THE UNIVERSITY
of EDINBURGH

Analysis and Transformation of Legacy Code

Stanislav Manilov

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2017

Abstract

Hardware evolves faster than software. While a hardware system might need replacement every one to five years, the average lifespan of a software system is a decade, with some instances living up to several decades. Inevitably, code outlives the platform it was developed for and may become legacy: development of the software stops, but maintenance has to continue to keep up with the evolving ecosystem. No new features are added, but the software is still used to fulfil its original purpose. Even in the cases where it is still functional (which discourages its replacement), legacy code is inefficient, costly to maintain, and a risk to security.

This thesis proposes methods to leverage the expertise put in the development of legacy code and to extend its useful lifespan, rather than to throw it away. A novel methodology is proposed, for automatically exploiting platform specific optimisations when retargeting a program to another platform. The key idea is to leverage the optimisation information embedded in vector processing intrinsic functions. The performance of the resulting code is shown to be close to the performance of manually retargeted programs, however with the human labour removed.

Building on top of that, the question of discovering optimisation information when there are no hints in the form of intrinsics or annotations is investigated. This thesis postulates that such information can potentially be extracted from profiling the data flow during executions of the program. A context-aware data dependence profiling system is described, detailing previously overlooked aspects in related research. The system is shown to be essential in surpassing the information that can be inferred statically, in particular about loop iterators.

Loop iterators are the controlling part of a loop. This thesis describes and evaluates a system for extracting the loop iterators in a program. It is found to significantly outperform previously known techniques and further increases the amount of information about the structure of a program that is available to a compiler. Combining this system with data dependence profiling improves its results even more. Loop iterator recognition enables other code modernising techniques, like source code rejuvenation and commutativity analysis. The former increases the use of idiomatic code and as a result increases the maintainability of the program. The latter can potentially drive parallelisation and thus dramatically improve runtime performance.

Acknowledgements

First and foremost, this thesis would not be possible without the tireless guidance and encouragement by my supervisor, Dr Björn Franke. By giving me the freedom to work independently he helped me develop my research skills, while at the same time he was always available and ready to discuss the next steps of my projects. The most important part of a PhD is the people you work with, and I was lucky to have a great supervisor. I would also like to thank Chris Vasiladiotis, without whom I would never manage to win against the quirks of the SPEC CPU2006 build system, but also for all the fruitful theoretical discussions. Also, Cedric Andrieu and Anthony Magrath from Cirrus Logic (the company generously providing the funding for my research) gave me an invaluable insight into the world of working for a processor manufacturer.

This section would be incomplete without mentioning all the friends I made at the Informatics Forum, Edinburgh University. Thanks to Akash Srivastava, Alejandro Bordallo, Alex Dawson, Craig McLaughlin, Daniel Hillerström, Emilian Radoi, Federico Pizutti, Harry Wagstaff, Hugh Leather, Janie Sinclair, Juan Fumero, Justs Zariņš, Kate Haag, Kiran Chandramohan, Luda Luisa Vissat, Maria Astefanoaei, Martin Rüfenacht, Nantas Nardelli, Paschalis Mpeis, Pavlos Petoumenos, Reese Stolzfus, Sam Ribeiro, Svetlin Penkov, Tom Spink, Ursula Chalita, Valentin Radu, and Yota Katsikouli, for creating the environment worth going to every day.

Special thanks to Andrew McLeod, Rui Li, and Kristian Ivanov for the emotional support during the most challenging times.

Lastly, thanks to Michel Steuwer for thoroughly reviewing the content of Chapter 4 and providing numerous suggestions on how to improve it.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Stanislav Manilov*)

To Marina, Zapren, and Nikola.

Table of Contents

1 Introduction **1**

1.1 Hypothesis 1
1.2 Introduction 2
1.3 Contributions 7
1.4 Structure 8
1.5 Publications 9

2 Free Rider
A Source-Level Transformation Tool for Retargeting Platform-Specific Intrinsic Functions **11**

2.1 Introduction 12
2.1.1 Motivating Example 13
2.1.2 Contributions 17
2.1.3 Overview 17
2.2 Background 18
2.2.1 Target Platforms 18
2.2.2 Benchmark Kernels and Application 20
2.3 FREE RIDER Methodology 20
2.3.1 Overview 20
2.3.2 Description of Intrinsics 22
2.3.3 Generation of C Header Files 27
2.3.4 Graph Matching and Source-level Transformation 30
2.3.5 Limitations 32
2.4 Empirical Evaluation 32
2.4.1 Evaluation Methodology 32
2.4.2 Benchmark Performance Results 35
2.4.3 Application Performance Results 36

2.4.4 Coverage and Frequency of Intrinsics 38
2.5 Related Work 41
2.6 Summary & Conclusions 43

3 Data Access Profiling **45**
3.1 Introduction 45
3.1.1 Contributions 47
3.1.2 Overview 47
3.2 Background 47
3.2.1 SPEC CPU2006 47
3.2.2 LLVM 49
3.3 Instrumentation 51
3.3.1 Memory Accesses 52
3.3.2 Loops 52
3.3.3 Function Calls 61
3.3.4 Putting it together 69
3.4 Profiling 69
3.4.1 Data Structures 69
3.4.2 Routines 72
3.5 Evaluation 73
3.5.1 Runtime 74
3.5.2 Memory Overhead 75
3.5.3 Choice of Input 76
3.6 Related Work 76
3.7 Summary and Conclusions 79

4 Generalized Loop Iterator Recognition **81**
4.1 Introduction 81
4.1.1 Motivating Examples and Use Cases 83
4.1.2 Contributions 85
4.1.3 Overview 87
4.2 Background 87
4.2.1 Affine Iterators 87
4.2.2 Induction Variable Recognition 88
4.2.3 RDS/DSWP Loop Partitioning 88
4.2.4 Object-oriented Iterators 89

4.2.5 Iteratorless and Inseparable Loops . . . 89
4.3 Methodology . . . 90
4.3.1 Definitions . . . 90
4.3.2 Static Analysis . . . 91
4.3.3 Incorporating Profiling Information . . . 92
4.3.4 Implementation . . . 94
4.4 Evaluation . . . 98
4.4.1 Experimental Set-up . . . 98
4.4.2 Results . . . 99
4.5 Related Work . . . 110
4.6 Further Analyses . . . 111
4.6.1 Decoupled Software Pipelining . . . 111
4.6.2 Code Rejuvenation . . . 112
4.6.3 Commutativity Analysis . . . 114
4.7 Summary, Conclusions & Future Work . . . 116

5 Summary **117**
5.1 Contributions . . . 117
5.1.1 Intrinsic Translation . . . 117
5.1.2 Data Dependence Profiling . . . 118
5.1.3 Iterator Recognition . . . 118
5.2 Critical Analysis and Further Work . . . 119
5.2.1 Limitations of Intrinsic Translation . . . 119
5.2.2 Performance of Data Dependence Profiling . . . 120
5.2.3 Specialisation of Iterator Recognition . . . 120

Bibliography **123**

Chapter 1

Introduction

1.1 Hypothesis

Executing legacy software on modern computer architectures is hampered by multiple issues. At the same time, until replacement systems are implemented – a non-trivial task worthy of consideration on its own – people, companies, and governments are forced to continue doing it. Following are three main challenges that come with running legacy software, together with techniques for addressing them.

First, some of the software – programs written to be optimised for an obsolete platform, in particular – cannot even be compiled for a new architecture. In such cases preliminary transformations need to be performed. Second, when programs can be compiled, analysing the source code only does not provide enough information for compiling them into efficiently executing binaries. Profiling and analysing the behaviour of example executions of these programs, i.e. applying dynamic analysis to them, dramatically increases the amount of information available to the compiler for making optimisation decisions. Third and last, once a program has been successfully compiled, and static and dynamic analysis have been performed, the resulting information still needs to be digested in a way that informs transformations aiming to optimise the efficient execution of the program. Focusing on program loops and separating each one of them into two parts – the collection of instructions that drive the execution of the loop and the collection of instructions that compute the information used later in the program – enables new approaches to transforming such loops.

These techniques for addressing the challenges that hamper the execution of legacy software on modern architectures are the focus of this thesis. Chapter 2 explores the preliminary transformations necessary for compiling platform-dependent legacy soft-

ware. Chapter 3 develops a dynamic analysis system specialised for extracting fine-detailed program information. Chapter 4 describes and evaluates an optimisation enabling loop separation analysis which combines static and dynamic program information.

1.2 Introduction

Hardware is evolving faster than software. The typical life span of a hardware system is between one and five years, while for software systems this is up to several decades[1]. As a consequence of this discrepancy, the execution platform for a given program can change to hardware systems it was not developed for. Even if a software system evolves functionally, it has been shown that with time it degrades structurally: its complexity increases, while its manageability decreases [13]. As a result, there is a point in time when it is judged more cost effective to stop the development of a system and to recreate it. Often, however, the old version would remain in use, at which point it becomes legacy.

There is no agreed-upon definition of what 'legacy code' means. For the purpose of this thesis, legacy code is a program (or a version of a program) that is no longer extended functionally but is still used for its original purpose. If there is any ongoing development work it is mostly done for maintenance or fixing security weaknesses. Ideally, legacy code would not exist and users would move on to the newest version of a program because that promises to provide the most up-to-date features, active support, and recent security upgrades. Practically, there are various reasons this is not the case and legacy software remains in use, including the financial cost of upgrading, the cost of retraining users due to major interface changes, the lack of compatibility of a newer version with other outdated elements of the hardware/software ecosystem, and others.

There are multiple issues with legacy code. Firstly, some systems are so old that there are no experts left who can maintain them. In late 2015, a Paris airport was brought to a standstill, because of a failure in its communication software. The software was so old that it was compatible only with Windows 3.1 [20] - a twenty-five-year-old operating system - and at the time there were only three people in France who could maintain that communication system. One of them retired soon after the incident.

[1] USA's air traffic control software infrastructure is reportedly forty years of age [19].

Legacy systems that do not suffer from such severe lack of maintenance experts, have another problem: the programming language used for their development can become outdated. Tools to compile and optimise the program on newer hardware might not be sufficiently mature and performance utilisation might suffer. The banking industry, for example, is notorious for its reluctance to update its software systems [29]. Even when senior managers recognize the problem the cost of modernisation is inhibiting as there are few tools to help with the endeavour. The work has to be performed manually and thus would take a long time and, inevitably, introduce numerous defects[2]. As a result, there are an estimated 220 billion lines of COBOL still in use today as part of various banking systems worldwide [41]. At the same time, COBOL has dropped in popularity in the past few decades [4] and there are few modern tools developed for it.

However, even if a legacy system is developed in a language that does not lose its popularity over time, there is a third problem. While there are cutting edge compilers and build systems capable of producing an application out of a legacy code base, these systems still need to analyse the structure of the source code if they are to produce efficient machine code that fully utilises the target hardware architecture. The rise of chip-level multiprocessing exacerbates this problem. Since 2005 the increase in processor frequencies has slowed down and the number of cores capable of parallel computing has started increasing instead (see Figure 1.1). In fact, single core performance of high-end chips has even started to drop in some cases, where system designers have focused on energy efficiency and multiprocessing performance. For such systems, and *without any additional analysis*, software designed for the single-core era will not execute faster or more efficiently and the hardware will be underutilised.

Although the emergence of the multicore era was largely unexpected[3], the computer industry was little concerned at the start. There were few indications that the end of frequency scaling would prove to be such an insurmountable issue for continuous hardware performance improvement. Crucially, however, the necessary compiler analyses needed to transform a general program, from a sequentially flowing string of logic and computations to an orchestra of communicating and collaborating, yet independent

[2] A rule of thumb in software engineering is that the number of errors per one thousand lines of code is between fifteen and fifty, regardless of the programming language used [60].

[3] Notably, the International Technology Roadmap for Semiconductors for 2005 and even 2007 was predicting continuing exponential growth of processor frequencies up to 2022 [1]. If that was the case there would be much less pressure on processor manufacturers to find alternative ways of delivering on the promise of performance growth, and the employment of multicore designs would have been optional, rather than the only option.

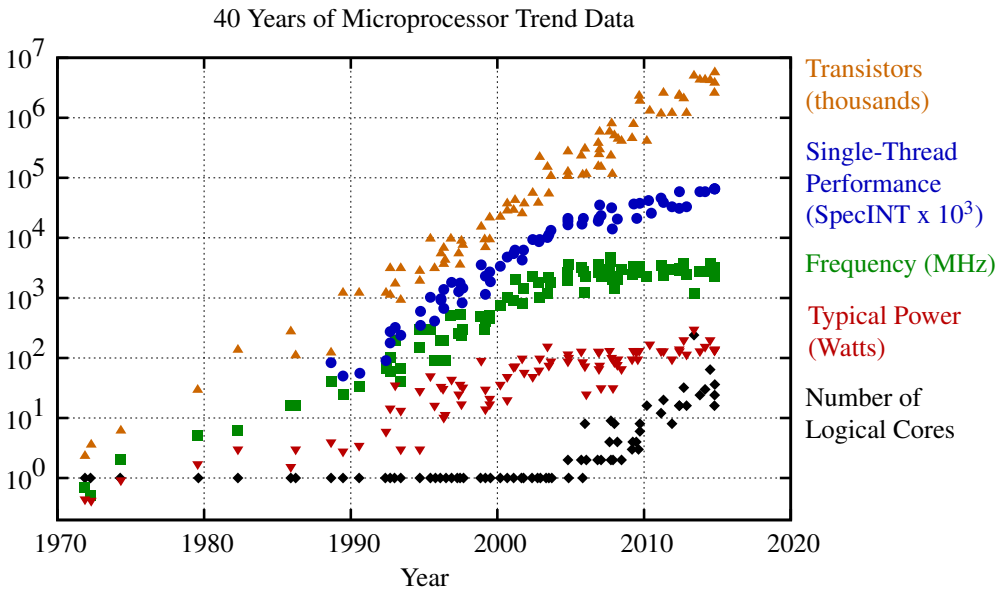


Figure 1.1: Comparison of key microprocessor characteristics over the last forty years (the data is taken from [87] whereas the graph is an original reproduction). The year of 2005 signifies the emergence of multicore processors while maximum frequency and single-core performance begins to stagnate.

execution threads, was discovered to be a formidable task.

The problem of this compiler transformation is known as automatic parallelisation. And while the research into this problem is almost as old as the first computers capable of parallel execution (see [53], e.g.) state-of-the-art industry compilers achieve no performance improvement due to automatic parallelisation, for the average program [92]. The reason for this failure is that while there have been promising speed improvements for specific types of programming constructs (for example simple loops that perform the same operation on each element of an array), in general, programs spend the most of their execution time in parts of the code which cannot be so simply described. A more abstract view of a program is necessary in such cases so that an analysis does not apply only to *for* loops, but to all loops, and that it does not apply only to array accesses, but to any kind of data structure accesses.

This leads to analysing legacy programs in a control flow graph (CFG) representation, and treating all data accesses equally by representing them as pointer dereferencing. If the semantics of the program is to be preserved by a parallelising transformation, the order of operations that depend on each other for their inputs needs to be maintained. This task, however, known as dependence analysis, has been proven to

be mathematically undecidable *statically* [54], i.e. by only analysing the source code and not inferring information from the execution of the program. For this reason, it is impossible for a perfect solution of automatic parallelisation that works for all programs to exist. Nevertheless, state-of-the-art automatically parallelising compilers still either analyse a very limited part of the code subject to compilation, or attempt to find a perfect solution to this impossible problem and fail.

The approach taken in this thesis is more pragmatic. Instead of attempting to start from a completely sequential program and produce a perfectly parallel equivalent – a task that is provably impossible in general – this thesis first focuses on programs that contain some information about their potential parallel execution, but they have that information expressed in a form that is not directly accessible for different hardware architectures, being target specific instead. This approach to improving the performance of a program is popular in signal processing: there, widely used audio and image processing libraries are written in a high-level language (like C or C++) but also make use of vector processing intrinsic functions to speed up their execution. These intrinsic functions are not implemented in the programming language of choice, but are rather instructing the compiler to generate specific high-performance processor instructions. As such, they are unique for each different instruction set architecture and make the program non-portable.

The technique described in Chapter 2 identifies intrinsic functions in programs and, leveraging the parallel execution information contained within, transforms the program to use identical or similar intrinsic functions in a different instruction set architecture: one that was not anticipated when the original source code was written. This is not always possible, so the functionality sometimes needs to be emulated and this is achieved by implementing the unmatched functionality in the high-level programming language. This emulation is only done when necessary, as it fails to translate the performance benefit of using the intrinsic functions with the original hardware platform. Nevertheless, the evaluation in Chapter 2 demonstrates that with the technique presented, performance close to that achieved by an expert manual translation can often be achieved. This means that the methodology can be applied to save large amounts of manual labour for all but the most performance critical applications.

Following Chapter 2, a larger problem is considered: what can be done when there is no parallel execution information available in the first place. As already mentioned, the problem is mathematically unsolvable. Still, that does not stop people from attempting to parallelise programs manually, and often succeeding. This fact leads to the

following insight: the main advantage human parallelisers have over compilers is that they do not restrict themselves to the source code only. People often exceed compilers in the complexity of their understanding of the structure of a program. That advantage comes from an understanding of the relationships between program modules, from an understanding of the algorithms involved, and from an understanding of properties of the data structures that are not explicit in the source code. A typical example is that of a sparse matrix data structure. Sparse matrices can be compactly expressed as a list of pairs: an index and a non-zero value. For a human it is obvious that no two elements will have the same index, but this (as well as many other restrictions on data structures) is often not expressed in source code. As a result, a compiler that analyses a loop iterating over the elements of a sparse matrix cannot *safely* assume that the index contained in these elements is going to be unique for each of them and thus can potentially miss a parallelisation opportunity, if it observes its requirement to maintain the semantics of the program.

A possible solution of this problem is to add the missing information in the source code or teach future programmers to specify such restrictions as part of the initial crafting of the program. This, however, inevitably requires manual effort which goes against the very objective of automatic parallelisation. Alternatively, a compiler should be allowed to infer this information independently of human guidance. The program input data can be identified as the source of that information missing from the program's code. Data access profiling has been used for several decades in order to gain insights from the program input data and the way it flows through the instructions[4]. However, research has mainly focused on runtime and memory performance optimisation. This has been achieved at the price of sacrificing robustness and precision. There is no data profiling framework that has been reported to be able to profile all of the SPEC CPU2006 benchmarks and that is able to track all dependencies between individual instructions.

Chapter 3 approaches the need for such a framework capturing the information about the flow of the data of a running program. A system tackling the data profiling problem is developed. It tracks data dependencies while identifying the execution context down to a loop-level granularity. The major contribution is carefully handling recursion and indirect function calls, which leads to the system achieving the specified goals of robustness and precision: it manages to profile all of the SPEC CPU2006

[4]Some early research efforts on using data dependence profiling to measure parallelisation are [52] and [55], while [23] are the first to focus on the problem of data dependence profiling exclusively.

benchmarks and builds an instruction-level dependency graph. As a result, the framework is used to build data access profiles crucial for the success of the methodology developed in Chapter 4. The runtime of the profiling framework developed in Chapter 3 is not as good as other state-of-the-art techniques, but it is not prohibitive and at the same time it manages to extract information from programs that other tools cannot analyse.

Chapter 4 focuses on the analysis of looping constructs in the CFG of programs. The goal is to cover as many different syntactic structures as possible under one definition of a loop. An analysis is developed, to separate these loops into an iterator (driving part) and payload (part doing useful work). The chapter shows that by focusing on this specific problem the analysis is able to extract this information for substantially more of these loops than previous state-of-the-art techniques. The success of the analysis is further extended by combining it with the data access profiling framework presented in Chapter 3.

Once a loop has been thus separated, further complex analyses can be performed on the two parts. One could analyse the loop iterator, looking for data structure access patterns, e.g. discovering that each element of a structure is accessed only once or in a particular order. Approaches like this can drive source code rejuvenation: the transformation of ad hoc syntactic structures into idiomatic code. Alternatively, commutativity analysis can be performed on the loop payload in order to experimentally discover whether the execution order of the loop iterations is part of the program semantics. If the order can be changed, then loop iterations can potentially be executed in parallel, given that access to shared memory is synchronised. These transformations are outside the scope of this thesis, but the analysis in Chapter 4 provides the critical information for their execution.

1.3 Contributions

The issues of legacy code can be summarised as two main problems:

1. functional portability: due to the use of non-standard language features or extensions, e.g. intrinsic functions, legacy programs sometimes cannot readily be compiled on a new hardware platform, even if there are mature programming tools for it;

2. performance portability: due to changes in hardware architectures – the advent of

the multicore era, in particular – legacy programs written for an old architecture rarely can automatically achieve the same amount of system utilization on a newer one.

This thesis approaches these problems via the following contributions:

1. A novel source-to-source compilation approach is described, for retargeting programs written in a high level imperative language that have been optimised through the use of platform-specific intrinsic functions. The main objective of this approach is to restore functional portability, but it is shown that a high-performance portability is also achieved, by reaching 96% of the performance of manually retargeted codes.

2. A framework for building data access profiles is introduced. The technique described in this thesis robustly handles the issues of recursive and indirect function calls when computing context identifiers and a set of formal definitions is presented, defining terms relating to loops, necessary for the accurate description of the instrumentation procedure.

3. The problem of performance portability is addressed by introducing a compiler analysis pass that detects generalised loop iterators: the part of a loop that is responsible for 'driving' the loop. This analysis is shown to detect considerably more loop iterators than the state-of-the-art scalar evolution analysis. This, however, is a statically undecidable problem in general, since it is based on dependence analysis. Thus, the analysis is further augmented by using data access profiling information collected by the profiling framework. This increases the detection rate from 53% to 81% for the SPEC CPU2006 C++ benchmarks and from 89% to 95% on average across all three languages (C, C++, and Fortran).

1.4 Structure

This thesis investigates modernising legacy code through the lens of compilation theory. Chapter 2 investigates the problem of restoring functional portability. In particular, it attacks the problem of retargeting a program written in a higher level language but having platform specific optimisations via intrinsic functions to another platform. The technique also achieves a reasonable performance portability by leveraging the optimisation information expressed by said intrinsics. When this information is missing,

optimisation opportunities need to be discovered. Chapter 3, builds the theory for, and describes, a robust context-aware data dependency profiling system. Such a system is needed in order to surpass static analysis during the effort of automatically analysing the structure of legacy software. Chapter 4 leverages the profiling system developed in Chapter 3 to develop a technique for recognising loop iterators. This technique can further enable different compiler analyses aiming to modernise code, for example source code rejuvenation, or commutativity analysis and, eventually, parallelisation. Chapter 5 summarises and finishes with a critical analysis of the contributions.

1.5 Publications

During the research of this thesis the following articles were published:

Generalized Profile-Guided Iterator Recognition. In *Proceedings of the 268th International Conference on Compiler Construction (CC)*, February 2018. Stanislav Manilov, Christos Vasiladiotis, Björn Franke. Extended here as Chapter 4.

Free Rider: A Source-Level Transformation Tool for Retargeting Platform-Specific Intrinsic Functions. In *ACM Transactions on Embedded Computing Systems (TECS)*, December 2016. Stanislav Manilov, Björn Franke, Anthony Magrath, and Cedric Andrieu. Presented here as Chapter 2.

Free Rider: A Tool for Retargeting Platform-Specific Intrinsic Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, June 2015. Stanislav Manilov, Björn Franke, Anthony Magrath, and Cedric Andrieu. Extended by the TECS'16 publication.

Chapter 3 is based on and extends a master's thesis [45]. The scope of extension is, as mentioned in Sections 1.3 and 1.4, a formal definition of terms related to loops for a more precise description of the instrumentation phase and a general solution to recursive functions and indirect function calls. In addition, Chapter 4 evaluates the profiler on all of the SPEC CPU2006 benchmarks.

Finally, contributions have been made to:

Towards a Compiler Analysis for Parallel Algorithmic Skeletons. In *Proceedings of the 268th International Conference on Compiler Construction (CC)*, February 2018. Tobias J.K. Edler von Koch, Stanislav Manilov, Christos Vasiladiotis, Murray Cole, Björn Franke. This publication presents a compiling technique which leverages the analysis presented in Chapter 4 and is discussed as further work in this thesis.

Chapter 2

Free Rider

A Source-Level Transformation Tool for Retargeting Platform-Specific Intrinsic Functions

Short-vector SIMD and DSP instructions are popular extensions to common ISAs. These extensions deliver excellent performance and compact code for some compute-intensive applications, but they require specialised compiler support. To enable the programmer to explicitly request the use of such an instruction, many C compilers provide platform-specific intrinsic functions, whose implementation is handled specially by the compiler. The use of such intrinsics, however, inevitably results in non-portable code. As the platform targeted by this optimisation becomes unavailable, code optimised in this way becomes legacy. This chapter describes a novel methodology for retargeting such non-portable code, which maps intrinsics from one platform to another, taking advantage of similar intrinsics on the target platform. A description language is employed, to specify the signature and semantics of intrinsics and perform graph-based pattern matching and high-level code transformations to derive optimised implementations exploiting the target's intrinsics, wherever possible. The effectiveness of the new methodology, implemented in the FREE RIDER tool, is demonstrated by automatically retargeting benchmarks derived from OPENCV sample programs and a complex embedded application optimised to run on an ARM CORTEX-M4 to an INTEL EDISON module with SSE4.2 instructions (and vice-versa). The tool achieves a speedup of up to 3.73 over a plain C baseline, and on average 96.0% of the speedup of manually ported and optimised versions of the benchmarks.

2.1 Introduction

Instruction set extensions are computer architects' favourite weapon of choice when adding domain-specific acceleration to processor cores with their mature and proven software and hardware ecosystems. For example, INTEL has devised various streaming SIMD extensions (first MMX, then SSE to SSE4 and AVX) to speed up graphics and digital signal processing. Similar capabilities are offered by the ALTIVEC floating point and integer SIMD extensions designed by APPLE, IBM and FREESCALE SEMICONDUCTOR. In the embedded space, ARM offers DSP and multimedia support through their SIMD extensions for multimedia and NEON extensions. Whilst conceptually similar, these different instruction set extensions differ significantly in detail, e.g. in their word and sub-word size, supported data types, and use of processor registers.

Despite improvements in compiler technology, including automatic vectorisation [72, 71], short-vector instructions offered by the architecture are typically accessed through platform-specific compiler *built-in functions*. This is due to the superior performance of hand-tuned vector code, which often outperforms auto-vectorised code [63]. Built-in functions, also called *intrinsics*, are functions available for use in C, but their implementation is handled specially in the compiler: the original intrinsic call is directly substituted by a machine instruction. For example, MICROSOFT's and INTEL's C/C++ compilers as well as GCC and LLVM implement intrinsics that map directly to the X86 SIMD instructions. The use of intrinsics enables programmers to exploit the underlying instruction set extensions and to increase the efficiency of their programs, but their use inevitably results in non-portable code. Obviously, this seriously restricts the re-use and porting of software components such as libraries, which have been heavily optimised for one particular instruction set extension and where no plain C sources are available.

In this chapter, based on a published journal article [59], a novel technique is developed, for cross-platform retargeting of code comprising platform-specific intrinsics. The **key idea** is to accept the presence of intrinsics as an opportunity and a source of information, rather than an obstacle. A graph based matching approach is developed, which aims at substituting existing intrinsics with those available on the target machine and possibly additional code providing compatibility. Descriptions of intrinsics are provided for a number of different instruction set extensions using a custom description language, covering the syntactic and semantic specification of intrinsics. These descriptions are translated to graph representations by the FREE RIDER tool, which

then translates any C program written using one set of intrinsics (e.g. those for an ARM CORTEX-M4 core) to make use of intrinsics of any other platform (e.g. INTEL SSE). Any pair of the available architectures can be used, in either direction. This translation process might also include additional source code transformations such as loop unrolling to account for different SIMD word sizes of the source and target platforms, respectively.

2.1.1 Motivating Example

Consider the example in Figure 2.1, which illustrates the steps involved in translating a vector addition loop using intrinsics for an ARM CORTEX-M4 to an INTEL SSE-enabled processor.

In Figure 2.1(a) ARM-optimised code is shown, which exploits the `UADD8` intrinsic available on the CORTEX-M4 platform and which provides convenient C-level access to a quad 8-bit unsigned addition instruction implemented in the processor's ISA. Using the `UADD8` intrinsic four pairs of one-byte values are added using a single processor instruction (line 10). To account for this implicit loop unrolling the surrounding loop is incremented in steps of four (in line 9), whilst also enabling 32-bit data accesses (rather than four individual 8-bit accesses). This is achieved by the access macro `PV`, which performs the necessary 32-bit cast operation. The measurable benefit of using the `UADD8` intrinsic in Figure 2.1(a) is a speedup of about four over a plain C implementation such as shown in Figure 2.1(b) (on a FREESCALE KINETIS K70 implementation of the ARM CORTEX-M4 core). However, higher performance for the platform-specific code comes at a price – the code in Figure 2.1(a) is **not portable** and does not work on platforms other than the ARM CORTEX-M4.

Porting of the code in Figure 2.1(a) to another platform is hindered by the fact that a plain C version such as shown in Figure 2.1(b) is often **not available**. In this situation, the user could (a) **manually derive** the plain C implementation and then try to vectorise this code, either manually or using an auto-vectoriser, or (b) use the FREE RIDER tool and methodology for **automatic retargeting**.

Now consider the automatically retargetted code, optimised for an INTEL processor with SSE extensions, in Figure 2.1(c). It exploits the `_mm_add_epi8` intrinsic, which provides access to an 8-bit addition instruction that operates on two groups of sixteen elements. Using the `_mm_add_epi8` intrinsic sixteen pairs of one-byte values are added in a single processor instruction (line 7). Accordingly, the loop increment has been

```c
char A[128], B[128], C[128];

// ... initialize A and B ...

// Packed vector access
#define PV(x) (*((uint32_t*)(&x)))

// Compute loop with UADD8 intrinsic
for (int i = 0; i < 128; i+=4) {
  PV(C[i])=__UADD8(PV(A[i]),PV(B[i]));
}

```

(a) Platform-specific code using the ARM `UADD8` intrinsic. This code cannot be compiled with a compiler, that does not support that intrinsic, and thus cannot be executed on non-ARM platforms.

```c
char A[128], B[128], C[128];

// ... initialize A and B ...

...
...

// Compute loop
for (int i = 0; i < 128; ++i) {
        C[i] = A[i] + B[i];
}

```

(b) Portable, but frequently **unavailable** plain-C implementation. Portable code versions are often not maintained or even dropped from code repositories as platform-specific optimizations are introduced.

```c
char A[128], B[128], C[128];

// ... initialize A and B ...

// Compute loop with Intel SSE intrinsic
for (int i = 0; i < 128; i+=16) {
        SV(C[i], _mm_add_epi8(LV(A[i]),
                              LV(B[i])));
}
```

(c) Platform-specific code using INTEL `_mm_add_epi8` intrinsic. Conceptually, the code looks similar to (a), but features a larger loop unrolling factor due to the target architecture's wider SIMD word size.

```c

// Load vector
#define LV(x) \
  (_mm_loadu_si128 ((__m128i*)(&x)))

// Store vector
#define SV(x, y) \
  (_mm_storeu_si128 ((__m128i*)(&x),y))

```

(d) Auxiliary load/store macros for INTEL SSE vectors complement the code in (c). On INTEL, vector accesses require special vector load and store intrinsics, whereas the ARM code in (a) only requires suitable casting.

Figure 2.1: Motivating example illustrating the use of the intrinsics to speed up a vector addition loop. The code in Figure 2.1(a) is optimised for an ARM CORTEX-M4. This code makes use of the ARM-specific `UADD8` intrinsic and **will not compile** for e.g. an INTEL platform. Equivalent plain-C code as shown in Figure 2.1(b) is often **not available**. Figure 2.1(c) shows the vector addition loop from Figure 2.1(a) **translated** to an INTEL platform, now using the INTEL `_mm_add_epi8` SSE intrinsic. This translation requires not only substitution of the ARM intrinsic, but additional code transformations. These comprise the introduction of suitable **short vector accesses** (Figure 2.1(d)), further **loop unrolling** to match the wider SIMD word size of the INTEL architecture and **dead store elimination** of redundant flag setting operations implicitly contained in the original ARM `UADD8` intrinsic, which are not used in this example, but need to be emulated where required.

adjusted to sixteen (line 6), and 128-bit data accesses are provided by the access macros `LV` and `SV`, shown in Figure 2.1(d). Without user intervention platform-specific ARM code has been retargetted to an INTEL platform whilst **retaining the performance benefit** of the original ARM intrinsic. Compared to a plain C baseline (such as the one in Figure 2.1(b)) the code in Figure 2.1(c) is about ten times faster[1].

FREE RIDER does not require a plain C implementation such as the one in Figure 2.1(b), but directly retargets platform-specific code *where no plain C implementation exists*. Translation of intrinsics involves a number of processing steps briefly outlined in Figure 2.2. We start with the code in Figure 2.1(a), but we do not have access to a plain C implementation such as the one shown in Figure 2.1(b). As a first step of the transformation process the `UADD8` intrinsic is expanded in the internal representation of FREE RIDER – it is essentially expressed as a vector of four additions followed by a vector of four compare-and-set operations. This is shown in Figure 2.2(b) and follows closely the specification of the `UADD8` intrinsic from Figure 2.2(a). The next step is to analyse which output of the intrinsic is actually used by the program. In the case of the motivating example the result of the addition is later used (for outputting the result, further computations, etc.), but the `APSR` register is never read. This register exists in the ARM CORTEX-M4 core to set flags indicating different program status - zero result, negative result, overflow (as is the case of `UADD8`), and others. Since the register is not read, writing to it is a waste of processing resources, so the compare-and-set operations are removed altogether. In general, a whole-program analysis is performed to check whether such writes to status registers are redundant or not. Another operation performed at this step is to find appropriate target SIMD intrinsic that consists of the remaining core operation – addition. In the case of INTEL SSE this is the `_mm_add_epi8` intrinsic, which has internal representation `(c = a + b) x 16` - it is a vector of sixteen additions. Since the widths of the vector operations do not match, the loop is unrolled to fit an `_mm_add_epi8` operation. This step is shown in Figure 2.2(c). Finally, when the resulting abstract representation matches exactly a target instruction it is replaced by that instruction together with appropriate access macros. The resulting code after retargeting, shown in Figure 2.2(d), matches the INTEL SSE implementation from Figure 2.1(d).

This methodology has been implemented in the FREE RIDER tool and its effectiveness is demonstrated using a set of compute-intensive OPENCV computer vision benchmarks [18]. Automatically retargeting these benchmarks from an ARM NEON

[1] Memory access overheads prevent the speedup to reach its ideal value of sixteen.

The __uadd8 intrinsic returns:

- the addition of the first bytes in each operand, in the first byte of the return value
- the addition of the second bytes in each operand, in the second byte of the return value
- the addition of the third bytes in each operand, in the third byte of the return value
- the addition of the fourth bytes in each operand, in the fourth byte of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If res is the return value, then:

- if res[7:0] $\geq$ 0x100 then APSR.GE[0] = 1 else 0
- if res[15:8] $\geq$ 0x100 then APSR.GE[1] = 1 else 0
- if res[23:16] $\geq$ 0x100 then APSR.GE[2] = 1 else 0
- if res[31:24] $\geq$ 0x100 then APSR.GE[3] = 1 else 0.

(a) Specification of the ARM CORTEX-M4 __uadd8 intrinsic [10].

```
1  // Compute loop -
2  // Abstract vector form
3
4
5  for (int i = 0; i < 128; i+=4)
6  {
7      vector_4 [j = 0 .. 3]
8      {
9          C[i+j] = A[i+j] + B[i+j];
10         APSR.GE[j] = (C[i+j] > 0x100) ? 1:0;
11     }
12 }
13
14
15
16
17
18
```

(b) Code in abstract vector representation.

```
1 // Reduced abstract representation
2 for (int i = 0; i < 128; i+= 16) {
3     (C[i] = A[i] + B[i]) x 16
4 }
```

(c) Unnecessary writes to APSR removed. Unroll factor increased.

```
1 // Compute loop - Intel SSE intrinsic
2 for (int i = 0; i < 128; i += 16) {
3   SV(C[i],_mm_add_epi8(LV(A[i]),LV(B[i])));
4 }
```

(d) Equivalent INTEL SSE code, which makes use of the _mm_add_epi8 intrinsic.

Figure 2.2: Motivating example illustrating the transformation from the use of the UADD8 intrinsic on the ARM CORTEX-M4 core (in Figure 2.1(a)) to the use of the INTEL _mm_add_epi8 SSE intrinsic (in Figure 2.1(c)). The specification of the source intrinsic (Figure 2.2(a)) is taken into account to convert the code to an abstract vector representation (Figure 2.2(b)). The transformation requires not only substitution of the ARM intrinsic, but additional code transformations, which comprise suitable short vector accesses and further loop unrolling to match the wider SIMD word size of the INTEL architecture. In addition, the overflow checking logic is removed, as the APSR register is not read later in the program, making the writes to it unnecessary (Figure 2.2(c)). Finally, the representation is lowered by using the target intrinsics (Figure 2.2(d)).

platform to an INTEL EDISON module with short-vector SSE4.2 instructions, achieves on average 96.0% of the performance of manually retargetted and optimised ports. Furthermore, an evaluation against a full-scale robotic application [61], which implements the computer vision component of a high-end autopilot for unmanned aerial vehicles (UAV) delivers a speedup of 3.73 over a plain C baseline, when ported from an ARM CORTEX-M4 platform to INTEL EDISON using the methodology presented here.

2.1.2 Contributions

This chapter makes the following contributions:

1. a novel, automated methodology for retargeting C code containing platform-specific intrinsics, whilst making efficient use of those intrinsics offered by the target platform is developed;

2. high-level descriptions of intrinsics, graph based matching and source-level code transformations to account for differences in the SIMD word sizes between machines are combined in the presented approach; and

3. the methodology is evaluated using compute-intensive OPENCV benchmarks as well as full applications and demonstrate performance levels competitive with manual retargeting efforts.

This chapter is based on a published journal article [59], which in turn extends an earlier conference article [58] in the following ways: a more complete specification of the FREE RIDER Description Language and the structure of the files generated by the FREE RIDER tool (in Section 2.3.3.1) is provided, the methodology is additionally evaluated and the OPENCV benchmarks are retargetted from INTEL SSE to ARM NEON to demonstrate both directions of translation, and the impact of the methodology on code size is discussed (in Section 2.4).

2.1.3 Overview

The remainder of this chapter is structured as follows. Section 2.2 introduces the background on compiler intrinsics, target platforms and applications. Section 2.3 presents the methodology for retargeting platform-specific intrinsics involving a high-level description of intrinsics, a graph-based matching algorithm, and source-level code transformations. The results of the evaluation on benchmarks and full applications are

presented in Section 2.4, before the context of related work is established in Section 2.5. Finally, Section 2.6 summarises and concludes.

2.2 Background

2.2.1 Target Platforms

The specific target platforms used in this research are the ARM CORTEX-M4 core, the INTEL X86 processor, with short-vector SSE4.2 instructions, and the ARM CORTEX-A9 processor, with ARM NEON SIMD instructions. The hardware implementations used for evaluation are an NXP KINETIS K70 MCU, an INTEL EDISON, and a PANDABOARD ES system, respectively. By providing further target descriptions (as described later in this chapter) other platforms such as POWERPC/ALTIVEC could be supported, but this is beyond the scope of this work.

The CORTEX-M4 processor is specifically developed to address digital signal control markets. It is designed so that it has low power consumption while offering high-efficiency signal processing functionality, provided by instruction set extensions accessible through ARM specific intrinsics.

ARM NEON is ARM's SIMD extension that targets more computationally demanding tasks, e.g. video processing, voice recognition, or computer graphics rendering. Architectures that support NEON have higher computational performance compared to the CORTEX-M4 processor and are typically found as application processors within mobile devices such as smartphones or tablets.

INTEL X86 on the other hand includes a huge family of processors, from embedded low-power chips to high-end server CPU's offering a one-size-fits-all instruction set. SSE4.2 is an instruction set extension that allows INTEL X86 processors to execute SIMD instructions on vectors up to 128-bit wide. This allows such processors to be used efficiently for multimedia and graphics processing.

SIMD operations, both for ARM and INTEL, are accessible to the C programmer by means of intrinsic functions. An intrinsic function is not explicitly defined by the programmer, but is provided (as a built-in function) by the compiler, which replaces an intrinsic function call with a hard-coded sequence of low-level instructions [12]. Examples for intrinsic functions are the `UADD8` intrinsic for the ARM CORTEX-M4 processor and the `_mm_add_epi8` intrinsic for the SSE instruction set extension, both of which are part of the motivating example (Figure 2.1).

In general, the operands of CORTEX-M4 SIMD instructions are 32-bit wide fields. Depending on the instruction each operand is treated as a single 32-bit number, two 16-bit numbers, or four 8-bit numbers. The available operations that can be performed range from simple operations like addition, to very specialised operations like the `SMLALDX` instruction which performs dual 16-bit exchange and multiply with addition of products and 64-bit accumulation. A list of the groups of available operations are: addition (13 instructions), subtraction (13 instructions), sum of absolute differences with or without accumulation (2 instructions), half-word multiply with addition or subtraction, with or without exchange, and with or without accumulation (12 instructions), parallel add and subtract half-words with exchange (12 instructions), sign-extend byte, with or without addition (4 instructions), half-word saturation (2 instructions), status register based selection (1 instruction).

At the same time, the operands of INTEL SSE4.2 instructions are 128-bit wide fields when they signify a vector, or any other type from the C programming language when they signify vector elements, bitmasks, or shift values. The 128-bit wide fields can be treated as vectors of two, four, eight, or sixteen elements of 64, 32, 16, or 8 bit values, respectively, depending on the instruction. The available operations are not as specialised as those of the CORTEX-M4 SIMD processor, but rather resemble standard processor instructions that operate on vectors instead of single elements.

While there are also miscellaneous utility (e.g. cache control) instructions for the INTEL SSE instruction set, primarily arithmetic instructions are targeted in this chapter. The integer instructions can be grouped in the following categories: addition (8 instructions), subtraction (8 instructions), sum of absolute differences (1 instruction), half-word multiply with addition (1 instruction), multiplication (5 instructions), maximum, minimum and average (6 instructions), shifts and bit-wise operations(22 instructions), comparison (9 instructions). The miscellaneous instructions that are of interest are shuffle instructions and pack/unpack instructions. They can be used to implement more complicated SIMD operations that include exchanging of vector elements.

Finally, the NEON extension is similar to SSE. Vectors can be either 64-bit or 128-bit wide and can contain signed or unsigned 8-bit, 16-bit, 32-bit, 64-bit integers, or single precision float numbers. The operations that are supported include standard arithmetic and logical operations, comparison operations, memory operations and shuffling operations. The more specialised operations which are not taken into consideration include table lookup and complicated mathematical operations, like reciprocal square-root estimate.

2.2.2 Benchmark Kernels and Application

There are no readily available standard benchmarks, which make explicit use of intrinsic functions due to the resulting undesirable restriction to a single platform. Therefore, a compute-intensive, open-source application extensively used in the academic, hobby and industrial communities is used for evaluation here. This application is PX4 [61] – a high-end autopilot for unmanned aerial vehicles (UAV) using computer vision algorithms – jointly developed by researchers from the Computer Vision and Geometry, the Autonomous Systems and the Automatic Control Labs at ETH Zurich (Swiss Federal Institute of Technology).

PX4 has been developed and optimised for an ARM CORTEX M4F CPU and, in particular, the optical flow module makes extensive use of SIMD intrinsics. Among the most computationally intensive functions in the computer vision component are those for the calculation of the Hessian matrix at a pixel location, the average pixel gradient of all horizontal and vertical steps, the Sum of Absolute Differences (SAD) of sub-pixel shift of two 8×8 pixel patterns, the SAD of two 8×8 pixel windows, and the pixel flow between two images. These functions have been extracted and used in isolation (to avoid system benchmarking involving the whole UAV) for the empirical evaluation. A single function (*absdiff*) is written entirely using ARM assembly, for which we provide a portable C implementation.

In addition, a number of benchmarks extracted from the popular OPENCV computer vision library [18] are used. These benchmarks comprise reference implementations, manually ported and optimised by an independent third party, supporting both ARM and INTEL through platform-specific intrinsics. The benchmarks are used to evaluate the performance and capabilities of the FREE RIDER retargeting tool in comparison to a manual effort.

2.3 FREE RIDER Methodology

2.3.1 Overview

The FREE RIDER tool performs four major transformation steps as shown in the overview diagram in Figure 2.3: header generation, data-flow extraction, graph matching, and source-level code transformation.

Initially descriptions of the source and target intrinsics are taken as inputs and emulation C header files (in the style of [101]) are generated. These header files declare

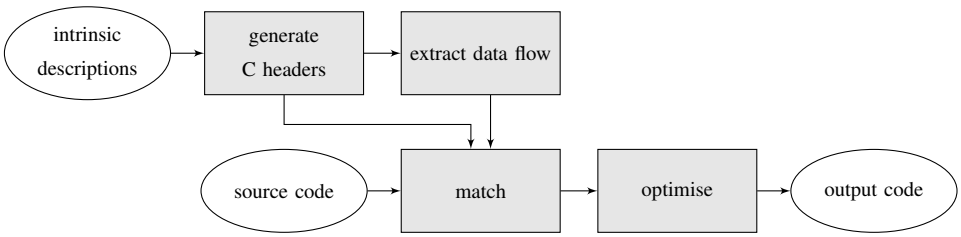


Figure 2.3: Stages of execution of the FREE RIDER tool

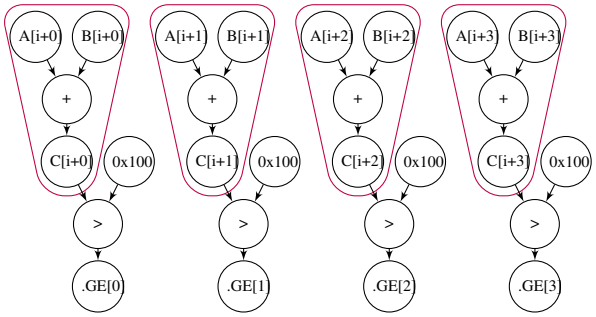


Figure 2.4: Graph representation of the `__UADD8` intrinsic. Arithmetic operations – circled in red – represent the transferable core of the vector add instruction, whereas the remaining parts implement the necessary comparisons for overflow checking and flag setting. Frequently, the flag setting operations can be eliminated if the flags are not used later on, i.e. they are “dead variables”.

and define portable C inline functions for the intrinsics of the source platform. Section 2.4 shows that the use of these "emulated" intrinsics results in portable code, but yields a low performance level of only 82% of a plain C implementation of the corresponding functionality on the target platform. This means that emulation of intrinsics through inline C functions provides compatibility, but results in a performance penalty.

In a second step the header files are used as input to the next stage, in which data flow graphs for each intrinsic are generated (see Figure 2.4 for example). These graphs, annotated with the types of inputs and outputs, serve as intermediate representation. Nodes of the graphs are also annotated with the operations performed, for example vector addition or vector sum reduction. These data flow graphs are later used for graph based pattern matching.

In the next step the C header files, the data flow graphs, and the source code of the program under consideration are all fed to the matching stage of FREE RIDER. It employs a greedy sub-graph isomorphism algorithm (similar to [57]) to match the data flow graph of each intrinsic encountered in the source code with data flow graphs of target intrinsics. The graphs of two target intrinsics can connect into a single graph,

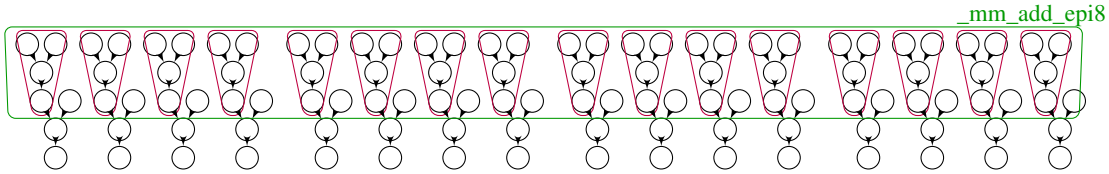


Figure 2.5: Matching of four __UADD8 intrinsics (in red) and one _mm_add_epi8 intrinsic (in green) resulting from sub-graph isomorphism detection, loop unrolling and dead variable/code elimination. The redundant flag setting computations have no counterpart in SSE and either require additional scalar C code or can be eliminated if there are no further uses of the flags (see also Figure 2.4).

by connecting the output nodes of one of them to the input nodes of the other by an assignment edge. In this way multiple target intrinsics can cover completely or partially a source intrinsic. For source intrinsics, which can only be partially covered by target intrinsics, scalar C code is generated for the remaining, non-covered parts of the data flow graphs. The output of this stage is C code of the target application with its source intrinsics partly or fully replaced with those of the target platform, wherever possible, and additional plain C code where a direct match is not possible.

Finally, the code resulting from substituting source intrinsics with target intrinsics is further optimised. Checks are performed to remove dead computations and variables (e.g. introduced as part of the flag setting operations in ARM intrinsics, see also Figure 2.5). Additionally, loop unrolling might be performed to adjust the possibly different SIMD word sizes of the two platforms (also shown in Figure 2.5).

2.3.2 Description of Intrinsics

Intrinsics are described in a high-level, human readable format. The description comprises the following items: name of native platform, list of operand names and types, output name and type, and the behaviour (a snippet of restricted C code). An abbreviated example of such a description is provided in Figure 2.6, which shows the specification of the ARM UADD8 intrinsic.

The platform declaration of an intrinsic describes the SIMD instruction set that is targeted, e.g. ARM NEON, ARM CORTEX M4, or INTEL SSE 4.2. Operand and result types can be standard C types, or vector types which should also be described in a format comprising the name of the native platform, total size in bits, and the type of a single element (atom type). While this allows for nested types of vectors of vectors,

```
define intrinsic UADD8
{
    platform ARM_CORTEX_M4
    operands val0:uint8x4_t@32, val1:uint8x4_t@32
    result   res:uint8x4_t@32
    behaviour {
        uint8x4_t res;

        // Load data and cast to prepare for
        // main operation
        uint16_t a0 = (uint16_t)UINT8X4_T_READ(val0,0);
        uint16_t a1 = (uint16_t)UINT8X4_T_READ(val0,1);
        uint16_t a2 = (uint16_t)UINT8X4_T_READ(val0,2);
        uint16_t a3 = (uint16_t)UINT8X4_T_READ(val0,3);

        uint16_t b0 = (uint16_t)UINT8X4_T_READ(val1,0);
        uint16_t b1 = (uint16_t)UINT8X4_T_READ(val1,1);
        uint16_t b2 = (uint16_t)UINT8X4_T_READ(val1,2);
        uint16_t b3 = (uint16_t)UINT8X4_T_READ(val1,3);

        // Perform additions
        // Need 16-bit intermediate results
        // to determine overflow flags
        uint16_t c0 = a0 + b0;
        uint16_t c1 = a1 + b1;
        uint16_t c2 = a2 + b2;
        uint16_t c3 = a3 + b3;

        // Assign results, casting to 8-bit
        UINT8X4_T_WRITE(res,0,(uint8_t)c0);
        UINT8X4_T_WRITE(res,1,(uint8_t)c1);
        UINT8X4_T_WRITE(res,2,(uint8_t)c2);
        UINT8X4_T_WRITE(res,3,(uint8_t)c3);

        // Flag setting, depending on
        // 16-bit intermediate results
        if (c0 >= 0x100) APSR_GE_SET(0); else APSR_GE_RESET(0);
        if (c1 >= 0x100) APSR_GE_SET(1); else APSR_GE_RESET(1);
        if (c2 >= 0x100) APSR_GE_SET(2); else APSR_GE_RESET(2);
        if (c3 >= 0x100) APSR_GE_SET(3); else APSR_GE_RESET(3);

        // Return result
        return res;
    }
}
```

Figure 2.6: Example showing the high-level description of the ARM `UADD8` intrinsic.

this feature is not used as there are no instructions that operate on such complex types. Thus, the atom type is a standard C type.

Behaviours of intrinsics, i.e. semantic actions, are expressed in a restricted subset of the C language. During generation of header files, behaviours are used as the function body of the generated inline function for the intrinsic.

Finally, platform-specific special registers can be described if they are used as part of the side effects of an intrinsic function execution. An example of such register is the ARM CORTEX-M4 `APSR` (Application Program Status Register), which is used e.g. for indicating arithmetic overflow.

2.3.2.1 FREE RIDER Description Language (FDL)

Figure 2.7 shows the FDL grammar in BNF form. As described earlier, the intrinsic definition includes a platform declaration, operand types, result type, and behaviour description.

The architecture, also called platform in the description, is specified by a unique identifying string, for example `ARM_CORTEX_M4`. As with the following constructs, an example of this is shown in Figure 2.6.

The types of the operands and the output of the intrinsic are either types defined in the `stdint.h` C header file (`uint8_t`, `int16_t`, etc.), or user defined types (shown later in Figure 2.9). Most of the time, the last would be used, as there are no standardized vector types in the C programming language, however the other two options are necessary for supporting particular operations e.g. operations involving accumulator variables. Alignment of the variables can be specified, by appending `@<align-size>` after the types.

The descriptions of the intrinsics are provided in a simplified code in the C programming language. The following guidelines have been observed during writing the behaviours for the experimental evaluation.

Behaviour descriptions should begin with reading of the separate elements of each of the argument vectors into separate variables. The names of these variables should be `a0`, `a1`, etc. for the elements of `val0`; `b0`, `b1`, etc. for the elements of `val1` and so on. Operations should then be performed on each pairs of elements consecutively, storing the results in a new variable every time. When all operations are performed, the results need to be stored in the different elements of the output vector `res`. Afterwards flag setting can be performed if needed and a single return statement should be present.

While this is a rather restrictive way of representing the behaviours of the intrinsics,

```
<fdl-file> ::= ("define" (<intrinsic> | <register> | <typedef>))*

<intrinsic> ::= "intrinsic" <name> "{"
                    "platform" <name>
                    "operands" <operands>
                    "result" <variable>
                    "behaviour" <behaviour>
                "}"
<operands> ::= <variable> ("," <variable>)*
<variable> ::= <name> ":" <name> "@" <number>
<name>     ::= "[a-zA-Z_][a-zA-Z0-9_]*"
<number>   ::= "[1-9][0-9]*"

<register> ::= "register" <name> "{"
                    "platform" <name>
                    "width" <number>
                    ("fields" "{" <field> ("," <field>)* "}")*
               "}"
<field> ::= <name> "[" <number> (".." <number>)? "]"

<typedef> ::= "type" <name> "{"
                    "platform" <name>
                    "bitwidth" <number>
                    "atomtype" <name>
                    "mapping" "(" "x" ":" "y" "[" <number> "]" ")" "{"
                          <bit-mapping>
                          <bit-mapping>*
                    "}"
              "}"
<bit-mapping> ::= "x" "[" <number> ".." <number> "]" "->"
                  "y" "[" <number> "]" "[" <number> ".." <number> "]"
```

Figure 2.7: The formal grammar of the FREE RIDER Description Language in BNF. A `<behaviour>` is a plain C function body and parsed using an embedded C parser.

```
define register APSR
{
    platform ARM_CORTEX_M4
    width    32
    fields   { N[31], Z[30], C[29], V[28], Q[27], GE[19..16] }
}
```

Figure 2.8: Example showing the high-level description of the ARM APSR register.

there are several advantages to it. Firstly, it is relatively easy to analyse the resulting C code and build a control-flow graph from it. Furthermore, due to the repetition of operations for each element, the graph can be manipulated to a control-flow graph of vector operations, rather than single element operations. At the same time the code representing the behaviour can be directly compiled by using any standard C compiler and thus no further modification of the code is needed to achieve portability. Furthermore, the format allows many default optimizations to be performed, which would eliminate the performance overhead of repetitive code. Finally, since the behaviour description is very explicit and does not include obscure statements involving multiple operations or potentially confusing pointer operations then it is easier to verify the correctness just by looking at the code and before using more detailed and robust verification mechanisms.

It is important to note that by no means are these restrictions imposed by the methodology, but are merely guidelines. The level of complication that is allowed in the behaviour description is limited only by the C language analysis sub-module of FREE RIDER, which is independent of the rest of the system and can be easily improved or replaced.

Moving on to defining a custom register, in order to do so one needs to specify its platform, bit width, and fields, each corresponding to a sequence of bits within the register. The platform declaration requirement is the same as for intrinsics and the bit width is a valid integer - usually a power of 2. Fields are specified as a list of named elements, each followed by its bit position within the register. If a field spans multiple bits then this can be specified by $\texttt{field}[\texttt{bit}_{\texttt{start}} \ldots \texttt{bit}_{\texttt{end}}]$. Figure 2.8 shows an example description of a register.

To conclude the section on FDL, platform specific vector types are defined using the type declaration construct. Users need to specify the platform, the total bit width

```
define type uint8x4_t
{
    platform ARM_CORTEX_M4
    bitwidth 32
    atomtype uint8_t
    mapping (x:y[4]) {
        x[7..0]       -> y[0][7..0]
        x[15..8]      -> y[1][7..0]
        x[23..16]     -> y[2][7..0]
        x[31..24]     -> y[3][7..0]
    }
}
```

Figure 2.9: Example showing the high-level description of the ARM type for a vector of 4, byte-sized unsigned integers.

of the vector, the type of a single element from it, and the mapping from bit ranges to vector elements. The platform declaration agrees with that of intrinsics and registers. The total bit width of the vector is the amount of memory that is required to store all of its elements. As an example, a vector of 8 elements, 8-bit wide each, has a total bit width of 64 bits.

The type of the single element of a vector type is called atom type in the description file. It must be one of the default types of the C programming language or a type defined in the `stdint.h` header file.

Finally, in the `mapping` field, one needs to specify the vector width as `w` in `y[w]`. It should agree with the total bit width divided by the width of the atom type. Each element `k` is mapped by a declaration of the form

$$\mathtt{x}[\mathtt{start_x} \ldots \mathtt{end_x}] \rightarrow \mathtt{y}[\mathtt{k}][\mathtt{start_y} \ldots \mathtt{end_y}]$$

where an index $= 0$ represents the least significant bit. This allows for conversions of endianness (by reversing the bit-range) and elements split over multiple locations. Figure 2.9 shows an example of a vector type definition.

2.3.3 Generation of C Header Files

After the intrinsic descriptions are provided they are used to generate one C header file per platform. Definitions of the custom types are output first together with macro

functions to read and write separate elements of a vector. Then, special registers are implemented as bit field structures and access macros for them are generated.

After this supporting code has been created, the implementation of the intrinsic functions as inline C functions is generated. Signatures are generated using the type information for the operands and the result, and the body of the functions are copied from the behaviour descriptions.

Using the generated header files, a data flow graph is derived for each intrinsic using standard data flow analysis techniques. These data flow graphs, together with the input/output type information of each intrinsic are used in the matching stage as descriptions of the intrinsics.

2.3.3.1 Structure of the generated files

Each generated header file describes all the information FREE RIDER has about the corresponding platform. Types are implemented as C `structs` that contain an array representing the vector. The type of the array is the same as the atom type in the definition, while its size is the total bitsize of the vector divided by the bitsize of an individual element.

In order to access the elements of a vector type `READ` and `WRITE` access macros are generated as part of the definition. These macros implement the switch of endiannes if there is one, and splitting and combining of fragmented elements, if there are any. In their simplest form, they just use the elements of the underlying C array as the elements of the vector. Figure 2.10 shows an example of a generated type.

```
1 typedef struct {
2     uint8_t _[4];
3 } uint8x4_t;
4
5 #define UINT8X4_T_READ(v, i) (v._[i])
6 #define UINT8X4_T_WRITE(v, i, x) (v._[i] = x)
```

Figure 2.10: Example showing the representation of the ARM type for a vector of 4, byte-sized unsigned integers. This representation is generated by FREE RIDER.

As mentioned earlier, platform specific registers are implemented as bitfields, with each flag given its required number of bits. Access macros are generated for each flag, so that they set and reset in an implementation independent manner. Flags that cover

multiple bits of the register have more complicated access macros, that take the index of the bit to be modified as an argument. Figure 2.11 shows an example of a generated register together with its access macros.

```
typedef struct {
    int _  : 23;
    int GE :  4;
    int Q  :  1;
    int V  :  1;
    int C  :  1;
    int Z  :  1;
    int N  :  1;
} _APSR;

_APSR APSR;

#define APSR_N_SET() (APSR.N = 1)
#define APSR_Z_SET() (APSR.Z = 1)
#define APSR_C_SET() (APSR.C = 1)
#define APSR_V_SET() (APSR.V = 1)
#define APSR_Q_SET() (APSR.Q = 1)
#define APSR_GE_SET(i) (APSR.GE |= (1 << i))

#define APSR_N_RESET() (APSR.N = 0)
#define APSR_Z_RESET() (APSR.Z = 0)
#define APSR_C_RESET() (APSR.C = 0)
#define APSR_V_RESET() (APSR.V = 0)
#define APSR_Q_RESET() (APSR.Q = 0)
#define APSR_GE_RESET(i) (APSR.GE &= ((1 << 4) - (1 << i) - 1)
   )
```

Figure 2.11: Example showing the representation of the ARM `APSR` register. This representation and access macros are generated by FREE RIDER.

Finally, the C functions implementing the intrinsics are generated. The return type is copied from the type of the `result` variable in the description, and the arguments have the types and names of the specified operands. The body of the function is directly pasted from the behaviour block in the description. Figure 2.12 shows an example of

a generated function emulating an intrinsic, with the body of the function omitted, as it is exactly the same as the one shown in 2.6.

```
1 uint8x4_t UADD8(uint8x4_t val0, uint8x4_t val1) {
2     ...
3 }
```

Figure 2.12: Example showing the implementation of the ARM `UADD8` intrinsic. The body of the function is equivalent with the behaviour block in Figure 2.6.

2.3.4 Graph Matching and Source-level Transformation

The process of matching intrinsics is outlined in Figure 2.13. Given the data flow graph of a source intrinsic, an intrinsic from the target platform is searched for using the VF2 graph/sub-graph isomorphism algorithm and library described in [24].

Since graph matching is an NP-complete problem, but one with great importance to many areas in computer science, there are multiple approaches to efficiently finding a solution. The one used for the FREE RIDER tool employs a deterministic algorithm and a State Space Representation (SSR) of the problem, to iteratively build a solution. There are five feasibility rules that are applied to pairs of nodes from the graphs being matched to help prune the search tree quickly. A total order relation guarantees that the applied Depth-First Search (DFS) does not reach the same state of the SSR twice, but via different paths. All of these techniques add up to an efficient graph-subgraph isomorphism algorithm. For more details, please refer to the original article [24].

The overhead that is added to the compilation time by using this algorithm is immeasurably small for all test cases. Its authors evaluate that it can match graphs up to 2000 nodes in under a tenth of a second. Since the graphs built by FREE RIDER used to represent the intrinsics are quite small in comparison (under 20 nodes for the most complicated intrinsics) the added overhead due to graph matching is negligible.

When a structural and operational match is found, the type information of the found operation is compared with the type information at the source location of the match. If the vector widths of the operands and the result match, the matching part of the graph is directly replaced with the found intrinsic and the process is repeated until no further unmatched parts of the source dataflow graph can be found or there are no more target operations that can cover the remaining graph.

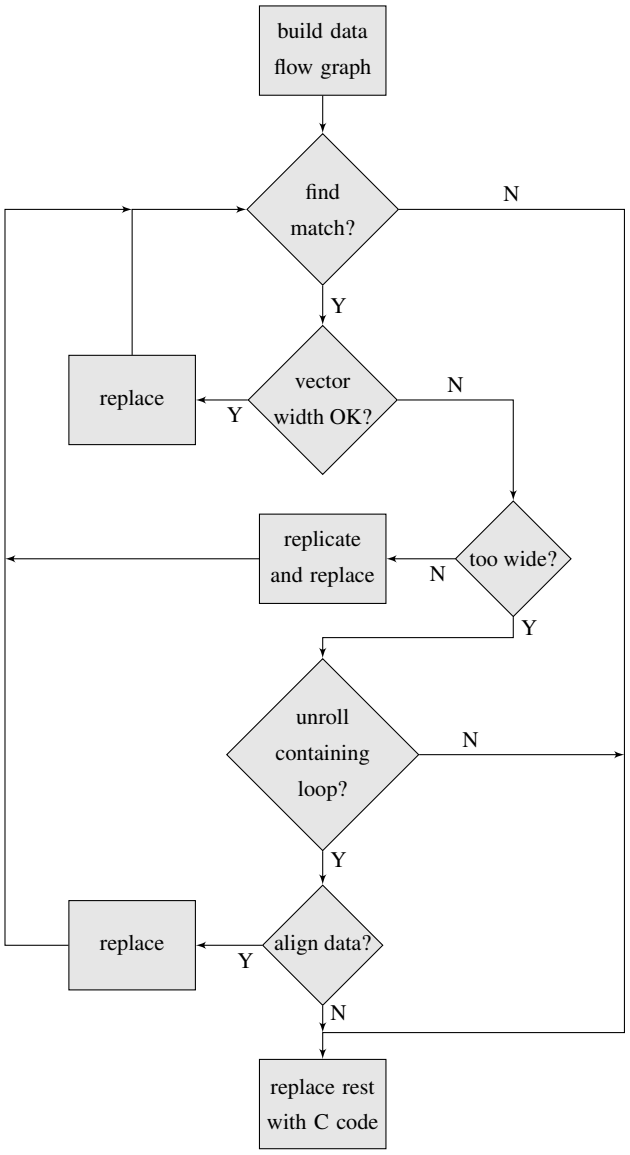


Figure 2.13: High-level algorithm for matching source and target intrinsics including loop unrolling for adjusting different SIMD word sizes and alignments. This algorithm is performed once per source intrinsic. In the case where the unrolling of the loop containing the intrinsic fails (e.g. when there is no such loop) then the matching of that intrinsic fails too. No further attempts are performed and instead, the intrinsic is emulated by replacing it with the equivalent C code.

There are two reasons for possible mismatches of vector widths: (a) The target intrinsic is too narrow (i.e. it contains fewer operands than the corresponding source intrinsic), or (b) it is too wide (i.e. it contains more operands than the source intrinsic). In the first case, the target intrinsic can be invoked as many times as it takes to match the width of the source vector (e.g. using four 4-element additions to implement one 16-element addition). In the second case, loop unrolling is required in order to enable a match (e.g. unroll a loop containing a 4-element addition in order to use a 16-element addition to implement four 4-element additions from four iterations).

In case loop unrolling is required it is performed alongside further data alignment. The latter might be necessary if arrays are not accessed in order, but some elements are skipped over. If either the unrolling or the data alignment steps fails, the matching fails and the default replacement with plain C code is performed to ensure correctness of the result. However, if they succeed the whole process is repeated again, until no further matches can be found.

Substitution of intrinsics as well as optimisation (loop unrolling, alignment, dead code/variable elimination) are implemented as source-level transformations. This means that C code enhanced with target intrinsics is generated, which can be compiled with the standard compiler for the target platform.

2.3.5 Limitations

As described in Section 2.2, intrinsics available for one platform cannot always be expressed by intrinsics available on another platform, or even in standard C code. Examples of such intrinsics are cache-ability and synchronisation operations. The approach presented in this chapter is limited to standard data processing operations and does not consider complex intrinsics whose behaviours cannot be expressed in C.

2.4 Empirical Evaluation

2.4.1 Evaluation Methodology

In order to evaluate it, the FREE RIDER methodology was applied to automatically retarget ARM-specific benchmarks and applications to an INTEL SSE enabled platform. The system used for performance evaluation is an INTEL EDISON module running at 500 MHz. The available physical memory is 1GB and the operating system is YOCTO LINUX, kernel version 3.10.17. All the benchmarks run on a single processor core.

In addition, INTEL SSE-optimised applications have been retargeted to an ARM NEON enabled platform. The system that was used for this part of the evaluation is a PANDABOARD ES device, containing an OMAP4460 system-on-chip that implements ARM CORTEX-A9 and running at 1.2 GHz. The available physical memory is 1GB and the operating system is UBUNTU LINUX, version 12.04. For this part, all benchmarks run on a single processor core too.

Performance is measured by using the UNIX program `time` to retrieve the total execution time of each benchmark. This is repeated between 30 and 100 times, depending on the variation of the measurements (until the 95 % confidence interval is 0.5 % around the mean). The average of all runtimes per benchmark is considered to be the representative runtime for that benchmark. Error bounds are not included, as they are too small to plot (less than 0.5 % for all benchmarks).

The OPENCV benchmarks are selected from the default sample programs that are provided with the OPENCV library, version `3.0.0-beta`. They have been prepared by removing the user interaction and substituting it with command line arguments and `stdout` messages. The eight benchmark programs selected for the evaluation of this chapter each contain a significant part ($>$ 10% of CPU time) executing vectorisable code. This was computed by compiling OPENCV with and without the included manual vector optimisations and comparing the runtimes of each benchmark for the two cases. Each of the benchmarks makes heavy use of functions provided by the OPENCV library. Many of these OPENCV functions have been manually ported and optimised for different target architectures, including ARM and INTEL. Benchmarks take between 0.3 and 100 seconds to execute, depending on the benchmark.

For the first part of the evaluation, the ARM ports of these functions are taken, they are automatically retargetted to INTEL, and then the performance of these automatically retargetted codes is evaluated in comparison to the manual INTEL port provided with OPENCV. For the second part, the opposite is done: the INTEL ports of the functions are taken, they are automatically retargetted to ARM-optimised code, and then their performance is evaluated in comparison to the manual ARM port provided with OPENCV.

Table 2.1 provides descriptions of the benchmarks. All of these benchmarks are real-world examples of programs from the computer vision domain.

The SSE intrinsics that were implemented include 14 arithmetic operations, 15 logical and comparison operations, 16 memory and initialisation operations, 4 conversion operations, and 8 shuffling operations. These correspond roughly to the NEON

Benchmark	Summary
calib	Calibrates a camera given a sequence of images.
bgfg	Split of background and foreground of video.
edge	Canny edge detection on an image.
align	Automatic alignment of an image.
polar	Polar transformation on a video.
segm	Automatic segmentation of objects in a video.
stitch	Stitching multiple images into a mosaic.
vstab	Automatic stabilisation of a video.

Table 2.1: OPENCV benchmark applications.

intrinsics that were implemented which include 21 arithmetic operations, 13 logical and comparison operations, 19 memory and initialisation operations, 12 conversion operations, and 8 shuffling operations. The greatest discrepancy is in the number of conversion operations. There are more NEON conversion operations because NEON allows for two vector widths (64- and 128-bit), and can convert between them, adding 8 operations.

The arithmetic operations comprise different versions of additions, subtractions, and multiplications, in addition to a single operation for division, maximum, minimum, and square root. The logical operations comprise different versions of logical ands, ors, xors, and shifts, whereas the comparison operations are comparisons for equality and strict inequalities.

The implemented memory operations comprise different loads and stores, whereas the initialisation operations are generating vectors, all depending on the data type of the given argument. The conversion operations convert the elements of the vector between different datatypes. Finally, the shuffling operations comprise instructions that reorder the elements of a vector in different ways.

On the target INTEL system, the CLANG/LLVM compiler is used to produce executable binaries. For comparison, an experiment where plain C sources are presented to the compiler for auto-vectorisation of the PX4 application was also conducted.

When targeting ARM, CLANG/LLVM was used for compilation again, this time however cross-compiling from an INTEL host.

2.4.2 Benchmark Performance Results

Figure 2.14 shows the results from running the automatically ported OPENCV ARM benchmarks on the INTEL evaluation system. While manually ported SSE code delivers a speedup of 1.26 over a plain C baseline, the FREE RIDER ports approach this performance delivering a speedup of 1.21. On average, FREE RIDER produced code that delivers a performance of about 96% of a manually ported and optimised INTEL SSE implementation. For every single benchmark the FREE RIDER port outperforms its plain C baseline, despite attempts of the compiler to auto-vectorise this plain C code. Even for the worst performing benchmark (`polar`) the automatically retargetted implementation outperforms the plain C baseline and delivers a speedup just about 6% lower than that of the manual port.

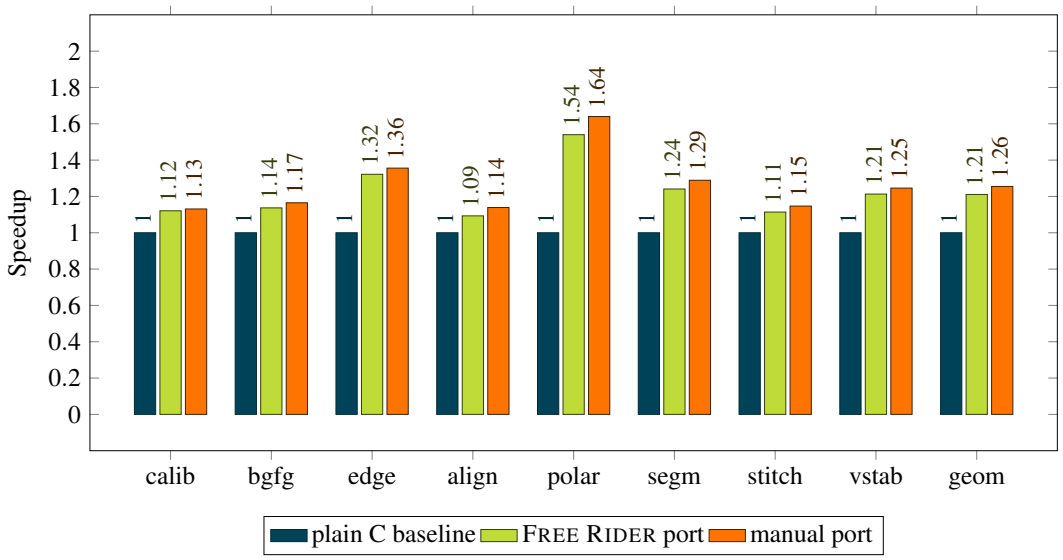


Figure 2.14: These OPENCV applications have been ported automatically from ARM NEON to INTEL SSE. Each bar reports the speedup per benchmark over a plain C baseline. In each case, the automatically retargetted version produced by FREE RIDER outperforms the plain C baseline. In fact, the performance of the auto-generated ports approaches that of the manually ported OPENCV applications, tuned extensively by the OPENCV community developers. On average, 96% of the speedup of a manual port is achieved, without paying the cost involved in manual code rewriting.

Similarly, Figure 2.15 shows the results from running the automatically ported OPENCV INTEL benchmarks on the ARM evaluation system. The manually optimized NEON code delivers a speedup of 1.31 over a plain C baseline, the FREE RIDER-ported applications perform comparatively well, with an average speedup of 1.26. Thus, FREE

RIDER produced code that again delivers a performance of about 96% of the manually ported and optimised ARM NEON implementation.

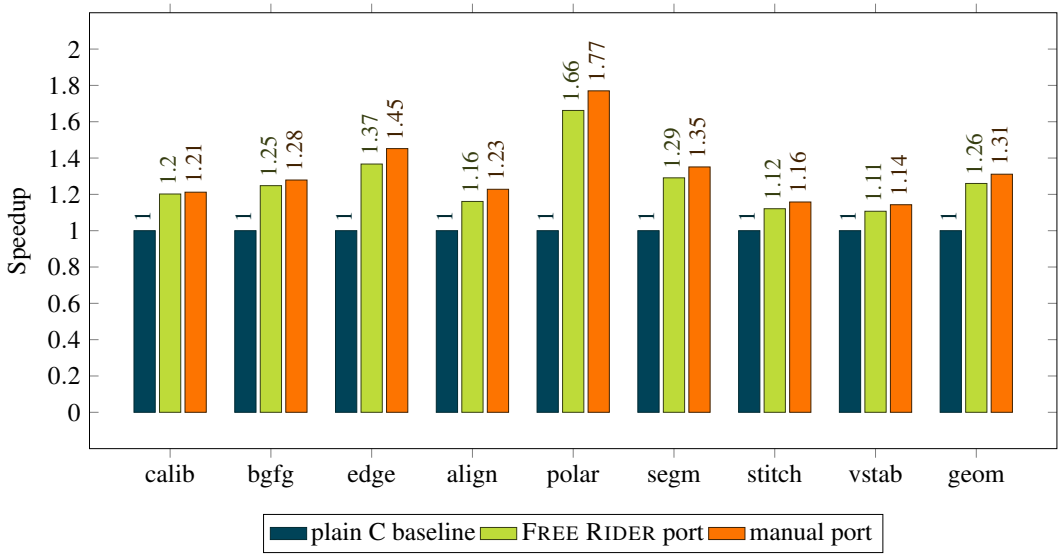


Figure 2.15: The same OPENCV applications as in Figure 2.14, this time ported automatically from INTEL SSE to ARM NEON. Expectedly, the results are similar to those shown in the previous Figure. 96% of the speedup of a manual port is still achieved, without paying the cost involved in manual code rewriting.

Closer inspection of the generated code revealed that the remaining performance differences between the manual and the auto-generated ports are mainly due to additional code restructuring performed by the expert programmers and optimisations to the scalar code surrounding the intrinsics.

For the evaluation of the impact on code size, the executable segments of the compiled OPENCV library were compared before and after optimization. It was observed that the difference between the two versions was negligible, with the optimized version less than 1% larger than before. The reason for this relatively small impact on code size is that the volume of code transformation is minuscule in comparison to the whole code base of the library. At the same time the resulting optimized code is contained in performance hotspots and contributes substantially to the overall runtime performance.

2.4.3 Application Performance Results

Next, let us focus on porting a larger application – the PX4 computer vision system – from ARM CORTEX-M4 to INTEL SSE. Figure 2.16 shows a set of results from running the target application in different configurations on the INTEL evaluation system.

The first configuration is a plain C baseline derived from the application's original sources. All further results are relative to this baseline configuration. As shown in Figure 2.16 the compiler fails to automatically vectorise this application, hence performance levels with and without compiler vectorisation are the same.

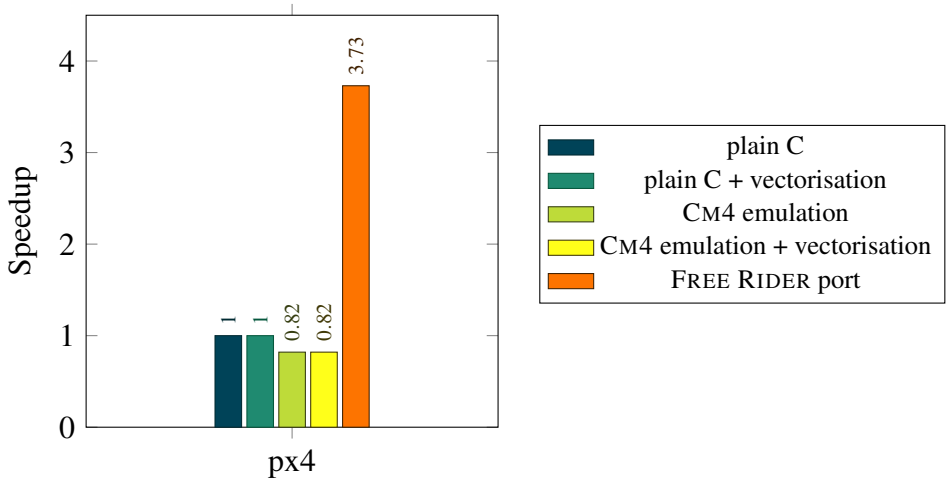


Figure 2.16: Relative performance of the ported PX4 application on the INTEL platform relative to a plain C baseline. Compiler vectorisation is not effective as it fails to detect and exploit vectorisation opportunities. Emulation of ARM intrinsics through inline functions, implemented in plain C, eases portability, but degrades performance with and without compiler vectorisation efforts. The automatically generated FREE RIDER port is capable of exploiting SIMD parallelism on the INTEL platform and delivers an almost four-fold speedup over the plain C baseline.

If ARM intrinsics are emulated by inline C functions, following an approach outlined in [101], performance suffers resulting in a drop of execution speed of about 18%. Again, the LLVM compiler's auto-vectoriser fails to exploit any vectorisation opportunities.

In contrast, the port produced by FREE RIDER substantially outperforms the baseline implementation. The reason for this is that even though FREE RIDER fails to exploit some optimisation opportunities due to irregularity of data accesses, it manages to vectorise the code in the most critical loop of the program and achieves a 3.73 performance speedup. It clearly demonstrates that FREE RIDER is capable of exploiting the source platform's intrinsics and mapping them onto corresponding intrinsics of the target platform. For the INTEL target platform used in this study this is close to the ideal four-fold speedup attainable using four-way SIMD processing.

Finally, Figure 2.17 illustrates how code size of the program is affected by the

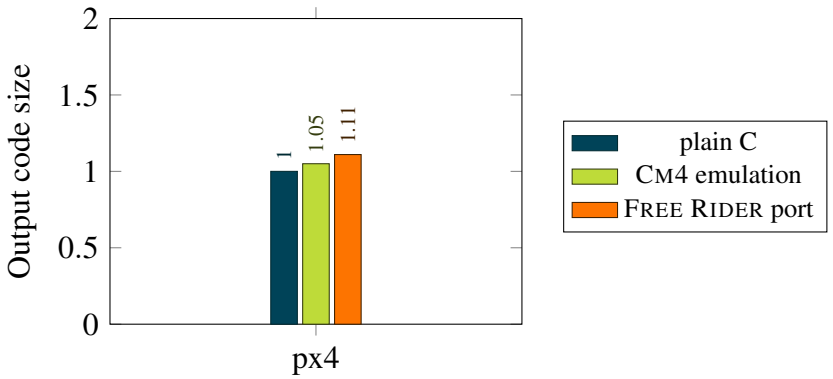


Figure 2.17: Relative code size of the ported PX4 application on the INTEL platform relative to a plain C baseline. In this example, the baseline application is optimized for code size and the emulation version is 5% larger. This is expected, as instructions which would include intrinsics in the original program are implemented as inline functions in the emulated one. The ported program is even larger, at 11% code size increase from the baseline. This is the case, because the translation of the intrinsics was produced after an aggressive transformation that results in multiple target intrinsics. This size growth is justified by the performance benefit discussed earlier.

transformation performed by FREE RIDER tool. The net effect is a 11% increase in code size, while performance increases by 273% over a plain C implementation. The increase in code size can be largely attributed to the source level transformations applied to the programme, in particular loop unrolling.

2.4.4 Coverage and Frequency of Intrinsics

Tables 2.2 and 2.3 list those intrinsics that were encountered in the translation process of the benchmarks (ARM NEON to INTEL SSE) and the larger application (ARM CORTEX-M4 to INTEL SSE). In addition, the frequency of occurrence in the benchmark sources is listed for each intrinsic. The first part of each table lists the intrinsics involved in the translation of the target application, while the second part lists the intrinsics involved in the translation of the benchmarks.

Some of the lines in the table represent multiple intrinsics, which is indicated by the names ending in an asterisk. An example is the `_mm_add*` line from the SSE table, which represents four intrinsics: `_mm_add_epi16`, `_mm_add_epi32`, `_mm_add_ps`, and `_mm_adds_epi16`. Also, some of the lines represent a group of instructions separated by a forward slash, for example `vceq*/vcgt*/vcge*/vlt*/vle*`.

Intrinsic	Frequency
__USADA8	36
__UHADD8	24
__UADD8	3
__USAD8	2
vld*	50
vadd*	38
vsh*	35
vdupq_n_*	27
vget_*	26
vmov*	26
vst*	24
vqmov*	22
vand*	16
vmul*	16
vcombine_*	15
vceq*/vcgt*/vcge*/vlt*/vle*	8
vsub*	7
vmin*/vmax*/vabs*	5
vqdmulhq_s16	4
vmla*	4
vbslq_f32	3
vzip_s16	2
vsqrt*	2

Table 2.2: Frequency of occurrence of ARM CORTEX-M4 and ARM NEON intrinsics in the benchmarks and application. Each line represents a single or multiple intrinsics. In the latter case the name ends with an asterisk to indicate that there are different variants available. The number of multiple intrinsics per line varies from 2 to 5.

Intrinsic	Frequency
_mm_srli_epi32	40
_mm_add_epi16	40
_mm_loadu_si128	34
_mm_store_si128	27
_mm_and_si128	24
_mm_set1_epi32	22
_mm_sub_epi16	20
_mm_abs_epi16	20
_mm_unpackhi_epi8	20
_mm_unpacklo_epi8	20
_mm_add_epi8	7
_mm_or_si128	6
_mm_load*	66
_mm_sll/sra/srl*	51
_mm_pack/unpack*	43
_mm_store*	40
_mm_add*	39
_mm_mul*	30
_mm_set*	27
_mm_and*	17
_mm_xor*	17
_mm_cmp*	16
_mm_cvt*	13
_mm_sub*	11
_mm_andnot_si128	7
_mm_sqrt_ps*	2
_mm_min/max_ps	2
_mm_div*	1
_mm_or_si128	1
_mm_movemask_epi8	1

Table 2.3: Frequency of occurrence of INTEL SSE intrinsics in the retargetted benchmarks and application. Asterisks are used similarly to Table 2.2. The first part of the table summarises statistics for the automatic translation of the target application, whereas the second part summarises statistics for the benchmarks.

The following a couple of interesting observations in Tables 2.2 and 2.3. Firstly, there are a lot more occurrences of INTEL intrinsics when compared to ARM intrinsics. This is because INTEL SSE was the destination platform during the experiments and a single source intrinsic might be mapped to one or more destination intrinsics. As a result, the code ends up with more target intrinsics than source intrinsics after the translation. This is more pronounced in the translation between ARM CORTEX-M4 and INTEL SSE compared to the translation between ARM NEON and INTEL SSE since the first pair of platforms is more dissimilar (as explained in Section 2.2.1) than the second. Secondly, the tables show that FREE RIDER is capable to cover wide range of intrinsics, which enable one to automatically process not only isolated benchmarks, but complex real-world applications resulting in performance levels approaching those of manual retargeting and optimisation.

2.5 Related Work

Handling of intrinsic functions by the compiler has found little attention in the academic community, possibly due to their normally straight-forward, but target- and compiler-specific implementation. Among the few publications dealing with various aspects related to intrinsic functions are the following.

Compilation for multimedia instructions has been an active research area for over a decade [89, 49, 78, 42, 90, 36]. Krall and Lelait [49] describe basic compilation techniques for multimedia processors. They compare classical vectorisation, borrowed from the age of the vector supercomputers, to using loop unrolling for vectorisation. The mentioned classical vectorisation employs a dependency analysis and might fail if the operations within the loop are not vectorisable. Loop unrolling is more likely to succeed, as the operations of consecutive loop iterations are the same – thus vectorisable. The only reason for failure is that there might be loop carried dependencies. The authors also explore the problem of unaligned memory accesses. FREE RIDER allows alignment to be specified in the description of architectures and honours it during translation.

Pokam et al. propose SWARP [78] – a retargetable preprocessor for multimedia instructions that is extendable by the user. Their work allows taking advantage of vector operations, without the programmer specifying that intention in the source code, i.e. the input provided to SWARP is plain C and it generates C code, which uses SIMD extensions. A flexible idiom recognition phase eases the retargeting of the system to

new machines without changing SWARP itself. The approach presented in this chapter is different in that it is retargeting platform-specific intrinsics. FREE RIDER leverages the expertise already invested in optimising the application to one platform and tries to maintain this information when translating to another platform. The idiom recognition is replaced by the matching phase of the technique presented in this chapter, and flexibility is achieved by the intrinsic description language, which is used to describe different targets.

Similar approaches, all operating on plain C input and trying to extract super-word level parallelism within an optimising or vectorising compiler are described in [90, 89, 42, 36]. Whilst these techniques are useful for the initial identification of vectorisation opportunities in C code, they fail to process applications, which have already been vectorised for a particular platform using intrinsics.

A graph based instruction selection technique has been developed in [70], where the compiler targets automatically generated instruction set extensions, where instruction patterns are not tree shaped, but highly irregular and sometimes larger (up to 12 inputs and 8 outputs) than typical multimedia instructions. The graph pattern matching approach used in this chapter is somewhat comparable to that in [70], however, the purpose of the work presented here is to aid the user retargeting an application optimised for a platform other than the current target platform, whereas graph pattern matching is used in [70] to match highly idiosyncratic instructions.

Modelling of instruction semantics in ADL processor descriptions for C compiler retargeting has been presented in [22]. The focus of this work is more on generating a basic compiler using an architecture description, rather than retargeting of existing, optimised code.

Implementation of intrinsic functions for a DSP compiler is subject of [12]. This article proposes and implements a new approach to intrinsic functions where the programmer targets a compiler's intermediate representation rather than the assembly language of a particular processor.

A general introduction to intrinsics for vector processing in the GCC compiler is provided in [48].

Possibly most relevant to the work presented in this chapter is [101], where a set of hand-coded inline functions compatible with ARM NEON intrinsics is provided for an INTEL platform with SSE. The result is a similar "emulation" layer providing portability for a particular combination of intrinsics (ARM NEON to INTEL SSE), but unlike FREE RIDER this is not automated and retargetable to any platform, but the result of a

major manual implementation effort for one specific pair of platforms.

2.6 Summary & Conclusions

This chapter has described a new methodology for retargeting platform-specific intrinsics from one platform to another. A description language is used to specify signatures and semantics of intrinsics of both platforms. These descriptions are processed by the FREE RIDER tool, which performs subgraph isomorphism checking to substitute one set of intrinsics with one or more intrinsics of the target platform, plus additional scalar code wherever needed. In addition, FREE RIDER performs source-level loop unrolling in order to account for differences in SIMD word sizes and alignment, and dead variable/code elimination to remove artefacts introduced by the substitution of intrinsics. The methodology has been evaluated by automatically porting OPENCV benchmarks optimised for ARM NEON and a compute-intensive application optimised for the ARM CORTEX-M4 processor to SSE4.2 enabled INTEL X86 processors (and vice-versa). This chapter demonstrates that FREE RIDER can take advantage of intrinsics, and that automatically retargetted code delivers performance levels comparable to manually optimised code for the target platform. A speedup of up to 3.73 over a plain C baseline is achieved on an INTEL EDISON module for the target application, and on average 96.0% of the speedup of manually ported and optimised versions of the benchmarks.

With this, a crucial aspect of the problem of **functional portability** has been addressed: usage of platform-specific intrinsic functions and how to overcome it when retargeting legacy programs to newer architectures. The approach presented in this chapter is also shown to maintain high enough **performance portability** so that the need for manual translation or performance tuning is alleviated except in the most performance critical applications. The main inspiration for this chapter (as the name of the project illustrates) is the leveraging of information about performance improvement opportunities that is *already present* in the program source code. The rest of the thesis focuses on *discovering* such opportunities instead, maximizing performance portability. Chapter 4 presents a novel generalised iterator recognition technique that combines static analysis and profiling information. Leading up to that, Chapter 3 presents an improvement of a recent context aware data profiling framework that is used to gather the necessary profiling information for the analysis in Chapter 4.

Chapter 3

Data Access Profiling

3.1 Introduction

Data dependence analysis for imperative languages is statically undecidable in the general case [54]. While it is possible to obtain concrete results for some limited cases, in general, if programs have calls to functions that cannot be inlined (e.g. external libraries) and perform pointer arithmetic with variables of unknown values (e.g. program inputs), then there might be dependencies where the source or target variables are impossible to pinpoint at compile time. Sophisticated static analyses can try to work around these obstacles and achieve practical (albeit partial) results, but even then, finding an example program that defeats these analyses is not a hard task.

A way to overcome the undecidability of static analysis is to use dynamic analysis: incorporating information about the program behaviour during runtime, in addition to the information that can be extracted from the source code statically. A dynamic system that tracks data dependencies is called a dependence profiling framework and consists of an instrumentation phase and a profiling phase. The subject program is first instrumented – calls to the profiling runtime are inserted at key points – and then it is profiled: the instrumented program is executed and runtime information is collected.

This chapter considers four features of dependence profiling frameworks: completeness, context precision, granularity, and runtime performance. A complete profiling framework can be used to profile any program written in the programming language it is made to handle, regardless of the way the language is used. While it is hard to quantify completeness, a framework that can be used to profile a superset of the programs that another framework can be used to profile can informally be said to be 'more complete'. The framework presented here is complete enough to analyse all of

the SPEC CPU2006 benchmarks. As such, it is strictly more complete than any other state-of-the-art dependence profiling framework.

Context precision concerns the amount of runtime information associated with a dependence, that the profiling framework collects. There could be no context at all, where only the static IR instructions which result in the dependence are recorded and in such cases there is zero context precision. When context is collected, however, it can include the function call stack at the point where an instruction is executed, loop nesting in the most recent function or inter-procedural loop nesting across the whole call stack, and even information about the state of data structures. This chapter presents a technique that collects call stack information and inter-procedural loop nesting information, as this is crucial information for its major application: the iterator recognition framework presented in Chapter 4. In order to achieve that, a new approach to updating the runtime context information during recursive and indirect function calls is presented here.

The granularity of profiling relates to the program elements that dependencies are built between. When it is fine-grained, profiling considers the smallest elements of the intermediate representation: instructions. The granularity spectrum goes through sets of instructions, whole basic blocks, groups of blocks, or entire function bodies. Most state-of-the art frameworks work above the instruction level granularity in order to achieve better runtime performance. In contrast, this framework tracks dependencies between instructions, as the desired outcome is a detailed dependence graph for every loop, needed by the iterator recognition framework presented in Chapter 4.

Finally, runtime performance is higher if the time taken to execute the instrumented program is shorter and the amount of memory used is smaller. Traditionally, research has been mainly concerned with reducing the runtime performance of profiling frameworks to the detriment of completeness, granularity, and context precision. The framework described in this chapter is complete enough to profile the SPEC CPU2006 benchmarks, has the context precision of inter-procedural loop nesting information combined with the call stack, and has instruction level granularity. At the same time, Section 3.5 shows that the runtime performance is good enough for the main application of the framework – enhancing the analysis presented in Chapter 4 – as profiling the SPEC CPU2006 benchmarks takes on average 7.1 hours and 9.4 GB of additional memory.

3.1.1 Contributions

This chapter presents a data dependence profiling framework. The main contributions of the work are the following:

1. The framework is strictly more complete than any other state-of-the-art framework as it is capable of profiling all of the SPEC CPU2006 benchmarks.

2. A new approach to updating the runtime context information during recursive and indirect function calls is presented, which allows efficient tracking of this information.

3. Evaluation shows that the performance of the technique is good enough for the main application of the framework: the one-off analysis presented in Chapter 4.

3.1.2 Overview

The rest of this chapter is structured as follows. Section 3.2 discusses the features of the LLVM compiler infrastructure, the intermediate representation, and the assembly language, after discussing the features of the SPEC CPU2006 benchmark suite. Section 3.3 builds a set of definitions and uses them to accurately describe the instrumentation phase of the profiling framework. Section 3.4 describes the functions and data structures that comprise the profiling runtime. Section 3.5 discusses the runtime performance of the framework, after which Section 3.6 presents the related work and Section 3.7 summarizes and concludes this chapter.

3.2 Background

3.2.1 SPEC CPU2006

The SPEC CPU2006 benchmark suite [39] was published in 2006 and replaced the previous CPU2000 suite. It contains 31 benchmarks that are written in three languages (C, C++, and Fortran), and are grouped in a set of integer benchmarks and a set of floating point benchmarks. The aim of SPEC is to provide honest data for the performance of computer systems including processors, memory subsystems, and compilers. For this reasons the benchmarks are drawn from real life applications, rather than using synthetic benchmarks and contrived programs.

3.2.1.1 Nature of Computation

The benchmarks in the CPU2006 suite cover a range of real life applications. The integer benchmarks include programming language implementations, artificial intelligence programs, discrete simulations, and compression algorithms. The floating point benchmarks include physics and chemistry simulations, mathematical solvers, a ray-tracer, and a speech recognition library. There is a special benchmark – specrand – which is not used for performance measurement, but rather as a fast test of system integrity: if it fails, the rest of the benchmarks are likely to fail too, as they include code from it. Excluding specrand[1], there are 29 benchmarks suitable for performance evaluation.

3.2.1.2 Parallelism

A study [7] of the speedup that can be achieved by compilers due to automatically detecting parallelism in the SPEC CPU2006 benchmarks, concludes that state-of-the-art industry compilers are highly incapable of utilising multi-core processors. Over the floating point set (the programs of which are more suitable for parallelisation), due to the many independent and regular computations that physics simulations comprise, the average speed-up due to parallelisation is reported to be 1.2x on a four-core machine. The result for the integer benchmarks is even more disheartening: the study reports no runtime improvement whatsoever over sequential execution.

3.2.1.3 Development

The CPU2006 benchmark suite was not updated for almost twice the life-span of the previous iteration: CPU2000. In June 2017, SPEC announced the release of CPU2017 [3], finally replacing the popular suite with a collection of modern programs. Unfortunately, this release was too late for the updated suite to be used for the evaluation of the techniques presented in this thesis. Nevertheless, this section summarizes some of the developments in the new iteration of SPEC CPU.

In addition to the integer/floating point divide, there is another dimension of categorisation introduced in SPEC CPU2017. Benchmarks are split into a ‘speed’ group: designed to measure runtime; and a ‘rate’ group: designed to measure throughput. The ‘speed’ group is similar to the way SPEC CPU2006 benchmarks are evaluated: one

[1] Technically, there are two versions of specrand, that is why the number of benchmarks is reduced from 31 to 29.

copy of the benchmark is executed and OpenMP can be used to speed up benchmarks that have been manually optimised to employ the multi-processing library. Benchmarks from the 'rate' group are evaluated by running multiple concurrent copies of the benchmark and OpenMP is disabled. In general, the benchmarks in 'speed' and 'rate' are derived from the same applications, although four programs are only available as 'rate' benchmarks and one is only available as a 'speed' benchmark.

Some programs from the CPU2006 suite have been retired and new ones have been introduced, but the general areas within the integer and floating point sets remain the same. The new programs in the integer set include more artificial intelligence algorithms and a different compression program. The new programs in the floating point set include more physics and weather simulations and different image rendering and manipulation tools.

3.2.2 LLVM

The LLVM compiler infrastructure [56] is an industry standard, extensible compiler library. It is open source and actively developed by contributors from a multitude of top computer industry companies including Apple, Google, Microsoft, Intel, Arm, and Qualcomm. It has also surpassed other compilers, both commercial and research ones, as the framework of choice when implementing prototype compiler analyses and transformations. As such, LLVM has become the de facto standard in compiler research too.

There are multiple advantages of LLVM that contribute to its popularity. The fact that it is open source and is freely available is one reason. More importantly from a technical perspective, however, is the modular design of the infrastructure and the unified intermediate representation. It is possible to combine different LLVM frontends, optimisations, and backends, and add support for languages which would otherwise require a tailor-made compiler. Alternatively, one can implement a single optimisation and combine it with the available tools in order to evaluate a novel idea without the overhead of implementing a frontend or a backend.

The profiling framework described in this chapter is implemented on top of LLVM (version 3.9). The rest of this section discusses important aspects of the infrastructure.

3.2.2.1 Optimisation Passes

The compilation process of LLVM proceeds in three stages: frontend, responsible for converting the high-level language source code into the intermediate representation (IR); optimisation, responsible for platform-independent analyses and transformations; and backend, responsible for lowering the IR to machine code. The whole process can be performed by a compiler driver (e.g. the clang tool) or it can be performed stage-by-stage while serialising the intermediate results to disc. This is enabled by the fact that the LLVM IR is designed to be serialisable as the LLVM assembly language.

While in theory it is irrelevant how optimisations are invoked, in practice the most convenient way is to configure the build system to generate a single LLVM module file just before invoking the assembler, process it with the desired optimisation pass, and then to assemble the result. This gives the optimisation a view of the whole program and allows it to analyse all functions encountered throughout the source code, as long as they are not implemented in dynamically linked libraries. The alternative would be to perform the optimisation on each of the compilation modules before linking and thus it would be impossible for certain information to be inferred, e.g. building a complete call graph.

In a setup where the desired optimisation is performed only once (by processing the module resulting from linking together all of the sub-modules of the program) this can be done using the `opt` tool provided by the LLVM project. Figure 3.1 shows an example compilation. This functionality allows for an optimisation to be developed outside the actual source code of the compiler and to be used as a plugin: simplifying further distribution and usage of the optimisation library.

```
1 clang <submodule-files> -c -emit-llvm -o preopt.bc
2 opt -load <opt-lib> -<opt-pass> preopt.bc -o <output-program>
```

Figure 3.1: Linking project sub-modules together and performing a custom whole program optimisation on the result. The `<opt-lib>` argument should be the library file that contains the implementation of the optimisation and `<opt-pass>` should be the name of the desired optimisation specified in the library.

3.2.2.2 Language

The LLVM assembly language is the textual representation of the LLVM IR. It is typed and in Static Single Assignment (SSA) form. A program is organised in modules which can be stored as separate files on disc, and each module is comprised of functions, global variables, and symbol table entries, alongside platform information and optional debugging information (named metadata). The code in functions resembles machine assembly code: it is organised in basic blocks of instructions, each having a single entry and exit points. Control between blocks is explicit and is implemented by terminator instructions.

The basic blocks of a function, together with the explicit control flow between them, form the Control Flow Graph (CFG) of the function. Instructions can be one of the following categories: terminator, binary, memory, or miscellaneous. The terminator instructions are control-flow changing instructions including branches and function returns. The binary instructions generally perform arithmetic computations including bitwise operations. Memory instructions are used to read and write from memory. (Memory locations in LLVM are not in SSA form, unlike the identifiers.) And the miscellaneous instructions include comparisons, function calls, phi instructions, and some special control flow instructions.

Although the LLVM language is designed to be human readable (and to some extent writable) this is done with the intention to aid debugging. It is not intended as a language for programs to be written in independently.

3.3 Instrumentation

The process of instrumenting a program involves inserting calls to a profiling runtime library at key points in the program. When the so modified program is then executed the profiling library collects information about the intermediate state and builds a profile.

The framework presented in this chapter inserts calls to indicate when memory is accessed: read or written; and calls to indicate context changes: loop entry/exit/iteration and function calls/returns in the original program. Although the process of instrumentation seems straight-forward at first, there are hurdles to be overcome before it can be successfully implemented, as illustrated by this section.

3.3.1 Memory Accesses

Instrumenting memory accesses is the trivial step of the instrumentation process. It consists of the following two rules:

- Before every memory store insert a call to **profile_store(memory_address, store_id)**.
- Before every memory load insert a call to **profile_load(memory_address, load_id)**.

The identifiers are assigned uniquely to loads and stores during a single pass of the whole program. This chapter assumes that all libraries of interest are linked at this point. Dynamically linked libraries will not be included in the profiling, which may result in incomplete information produced by the profiler. Incorporating dynamically linked libraries into the profiling is left as future work.

Note that memory access size is not included in the profiling. This is based on the assumption that the type of the variable stored in a given memory location does not change over its lifetime. When this assumption does not hold, e.g. when a binary subpart of a variable can be accessed as in a *union* from the C language, then some profiling precision will be lost. However, the rest of the extracted profiling information will still be correct.

3.3.2 Loops

It is important to precisely specify what is meant by a loop before the mechanism of instrumenting one can be described. This section builds a vocabulary of terms - entry and exit; start, restart and finish; iteration and traversal - in order to make explanations in the context of control flow changes easier. Only then are the profiling calls that deal with loop control changes specified.

3.3.2.1 Background

Traditionally, compiler analysis theory has been focusing on loops such as Fortran 'do-loops' (see [53] and [11]). For programmers in the past several decades the concept is usually introduced by syntactic constructs like 'for' and 'while' in newer languages, like C, C++, Java, Python, etc. As such, loops in programs usually arise from the use of one or more of these constructs.

Here, however, we work towards the definition of a loop as a recognisable object on the control-flow graph (CFG) level, that encompasses all such intuitions and more.

In particular, the goal is to obtain a single definition of the notion of a **loop** which deals with the majority of the CFG structures that can lead to the repetitive execution of the same code. The definition here is based on [6] and is compared to the one in [67].

First, recall some definitions from [6].

Definition 1. Let F be a flow graph and S a subset of blocks of F. We say that S is a **strongly connected region**(**region** for short) of F if:

1. There is a unique block B of S (the **entry**) such that there is a path from the begin node of F to B which does not pass through any other block in S.

2. There is a path (of nonzero length) that is wholly within S from every block in S to every other block in S.

Furthermore, a region is **maximal** if it contains every other region that has the same entry (see [6], Example 11.34).

[6] do not provide a concrete definition of what a loop is, instead only mentioning 'generalized loops in flow graphs are called "strongly connected regions".' This is not a sufficient definition, since it allows for the construction of partial loops and multiple distinct loops with the same entry. For the purposes of this thesis, a **loop** shall be defined as a **maximal region**, which is a more narrow definition than that of a 'generalized loop' (see Figure 3.2). Loops defined in this way are always either disjoint or completely contained in each other and never share their entry blocks.

Note that the notion of a strongly connected region is more narrow than the notion of **strongly connected components** (SCC) of a graph, where there is no requirement for the uniqueness of the entry node. If there is an SCC in the CFG of a function that has more than one entry node, then the CFG of that function is said to be **irreducible**. While these do appear in practice, a cursory check shows that they comprise 0.02% of the functions found in the SPEC CPU2006 benchmark suite. Luckily, despite the name, irreducible flow graphs (IFG) can be turned into reducible ones by applying a standard transformation which duplicates some of the blocks of the IFG (again, see [6], Section 11.4.3). Once a CFG is reducible all of its SCCs will have unique entries and thus will belong to a loop as defined in this chapter.

Now, let us compare this with the definition of a loop in [67]. It is based on the notion of *dominance*.

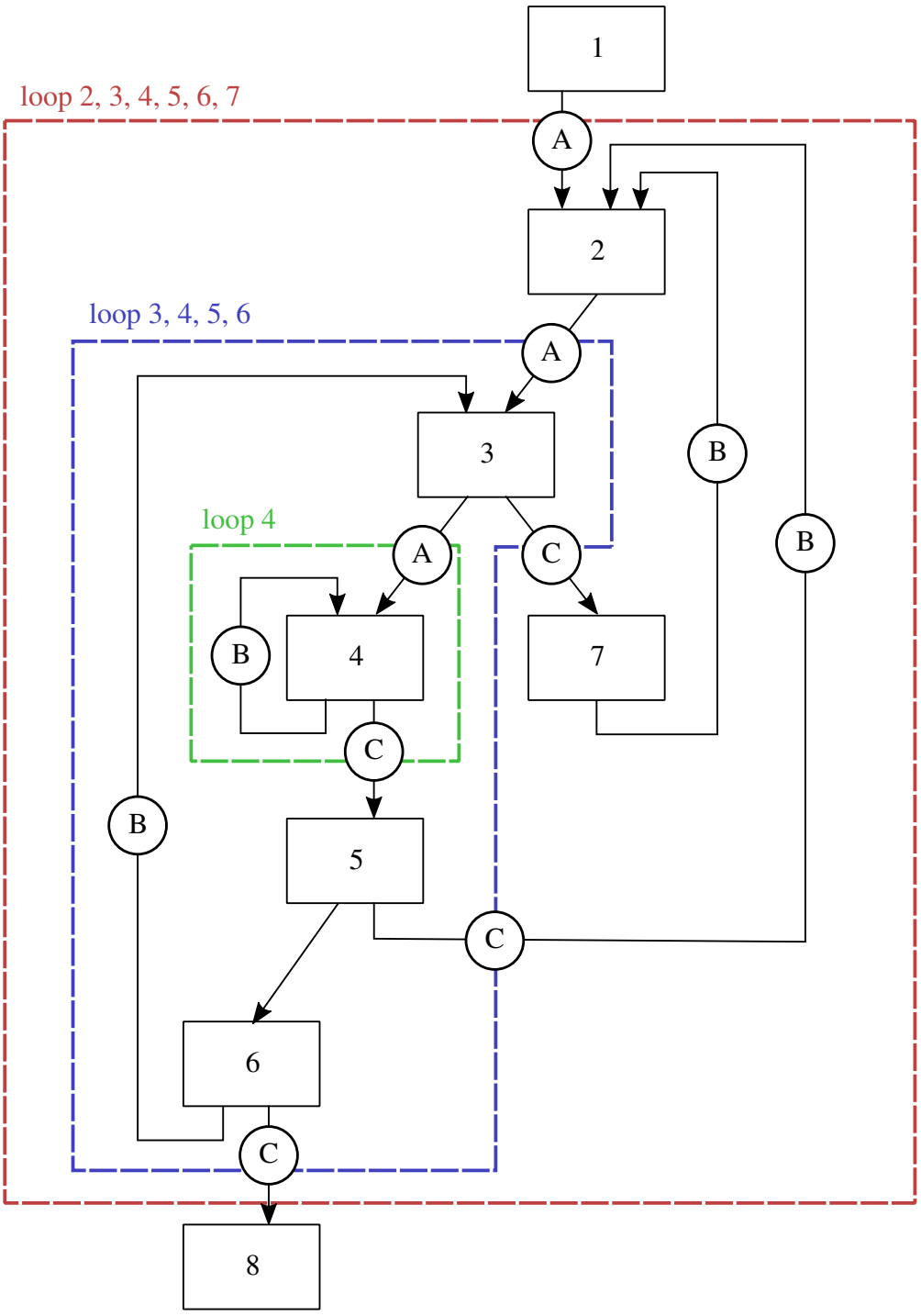


Figure 3.2: An example CFG from [6] and the loops found in it. In addition to the outermost loop (here marked with a red dashed outline) there are four more 'generalized loops' that have block 2 as their entry: {2,3,7}, {2,3,4,5}, {2,3,4,5,6}, and {2,3,4,5,7}. This makes them five in total. ([6] claim there are three in total which is a mistake. See [6], page 924). For the purposes of this thesis, there is a single loop with entry 2 - the corresponding maximal region. Two more loops can be identified: 3,4,5,6 and 4. The edges marked as A are loop starts, B are loop restarts, and C are loop finishes.

Definition 2. Let F be a flow graph and B and C be two blocks of F. We say that B dominates C if every possible execution path from the begin node of F to to C includes B.

[67] uses this to define the term *back edge* as 'one whose head dominates its tail' and builds his definition of a 'natural loop'[2] on top of that.

Definition 3. Let F be a flow graph and M and N be two blocks of F such that $M \rightarrow N$ is a back edge. The *natural loop* of $M \rightarrow N$ is the sub-graph consisting of the set of nodes containing N and all the nodes in F from which M can be reached without passing through N and the edge set connecting all the nodes in its node set.

Defining a loop by its back edges is not the most intuitive way to go. Even then, it is not clear what would the result be if a loop has multiple back edges. Indeed, by this definition natural loops can share an entry block, which is misleading since it can lead to a situation where execution alternates between the two loops without either of them terminating in the meantime. The notion of a loop presented here is based on maximal strongly connected regions infers a single loop entity and avoids this problem.

3.3.2.2 Terminology

For convenience, the earlier definition of a loop is repeated here.

Definition 4. Let F be a flow graph and L a subgraph of F. We say that L is a **loop** in F if:

1. There is a unique block B of L (the **entry**) such that there is a path from the begin node of F to B which does not pass through any other block in L.

2. There is a path (of nonzero length) that is wholly within L from every block in L to every block in L.

3. L is maximal: adding more blocks and edges from F to L will invalidate either 1 or 2.

The following is a definition of a loop **start**, which is intuitively an edge in the CFG which leads to the loop is:

[2] [67] does not specify what other loops there might be, but uses the term as a synonym for a cycle in the CFG.

Definition 5. Let L be a loop and E be the entry of L. If P is a block that is not in L and there is an edge from P to E, then we call that edge a **start** of L.

Example 1. In Figure 3.2, the edges marked as (A) are loop **start**s.

Next, loop **exit** and loop **finish** are defined. Intuitively, a loop **exit** is the last block of the loop to be executed and a loop **finish** is the edge in the CFG that leads from the loop **exit** to outside the loop.

Definition 6. Let L be a loop. An **exit** of L is a block B such that B precedes at least one block N that is not in L. We call the edge from B to N a **finish** of L.

Example 2. In Figure 3.2, nodes 3 and 6 are **exit**s for loop {3,4,5,6}, node 4 is an **exit** for loop {4}, node 6 is an **exit** for loop {2,3,4,5,6,7}, and the edges marked as (C) are loop **finish**es.

Next, **loop traversal** is defined:

Definition 7. Let L be a loop of the flow graph F and T a sub-graph of F such that the nodes of T are a subset of L and the edges of T are all edges in L that connect the nodes in T. We say that T is a **traversal** of L if:

1. T contains the entry E of L (we call E the **beginning** of T);
2. There is a unique node H in T such that there is an edge in T from H to E. H might be an exit of L, but it need not be. We call H the **end** of the traversal T. We also call the edge from H to E the **restart** of T (also, a **restart** of L); and
3. There is a path that is wholly within T from every block in T to every other block in T.

For clarity, **traversal**s will be identified by their set of nodes, but note that they contain all edges connecting these nodes in L too, since they are sub-graphs of a CFG by definition.

Example 3. In Figure 3.2 the traversals of loop $L = \{2,3,4,5,6,7\}$ are $T_1 = \{2,3,7\}$, $T_2 = \{2,3,4,5\}$, $T_3 = \{2,3,4,5,6\}$, and $T_4 = \{2,3,4,5,6,7\}$. Their corresponding **end**s are 7, 5, 5, and 7. Note that 5 cannot be the **end** of T_4, because due to uniqueness of ends (see Definition 7, Item 2) there would be no path from 7 to any other node of T_4 if it was.

To make an intuitive sense of the definition of a **traversal** it helps to consider a loop **iteration**:

Definition 8. Let T be a traversal of a loop L and let S be a (possibly repeating) sequence of nodes in T. We call S an **iteration of** L **associated with** T if:

1. S contains each node of T at least once;
2. S contains the start of T exactly once - as its first element;
3. The last element of S is the **end** of T;
4. If block B is followed by block C in S, then there is an edge from B to C in T.

Example 4. Continuing Example 3, some iterations of L are:

1. associated with T_1: only $I_{11} = \{2,3,7\}$;
2. associated with T_2: $I_{21} = \{2,3,4,5\}$ and $I_{22} = \{2,3,4,4,4,5\}$;
3. associated with T_3: $I_{31} = \{2,3,4,5,6,3,4,5\}$ and $I_{32} = \{2,3,4,5,6,3,4,4,5\}$ and $I_{33} = \{2,3,4,4,5,6,3,4,4,4,5,6,3,4,5\}$;
4. associated with T_4: $I_{41} = \{2,3,4,5,6,3,7\}$ and $I_{42} = \{2,3,4,5,6,3,4,5,6,3,7\}$.

Informally, a **traversal** of a loop is a subgraph of blocks that might be executed during any **iteration** of the loop in an execution of a program. Note that while a traversal is not a multi-set (the elements of the set are unique), a loop iteration might repeatedly execute the same block an arbitrary amount of times before terminating - usually when there is a nested loop, as defined in Definition 9.

Later, in Chapter 4, the notion of a *loop iterator* is defined. While the names of the terms loop iterator and loop iteration are very similar, the concepts are notably different. Intuitively, a loop iterator is the collection of variables and the operations performed on these variables that determine the execution of a loop, i.e. when different loop restarts or finishes are taken. These could comprise a part, or all of the loop. In contrast, a loop iteration is all of the operations executed at runtime between a loop start and a loop finish or restart. There could be multiple different loop iterations for a given loop while there is a unique loop iterator. Some part of an iteration necessarily is a part of the loop iterator, while the loop iterator does not have to contain the whole of any of the iterations. For more details about loop iterators see Chapter 4.

Definition 9. A loop K is **nested** in a loop L if K is completely contained in L. Furthermore, K is **directly nested** in L if for any iteration of L the *start* of K is encountered only once. Conversely, if there is an iteration of L that follows the start of K more than once there must exist another loop M, such that M is directly nested in L and K is nested in M.

To summarize, based on [6], a **loop** was defined as a **maximal region**. A definition for a loop **exit** was provided to complement the traditional definition of a loop **entry**. Loop **start**, **restart**, and **finish** were defined in order to simplify explanations in the context of control flow changes. A **traversal** of a loop was also defined, together with its **beginning**, **end**, and **restart**, and based on that a loop **iteration** was defined. Finally, the notions of a loop, a loop start, and a loop iteration are combined to define **nested** and **directly nested** loops.

3.3.2.3 Instrumentation

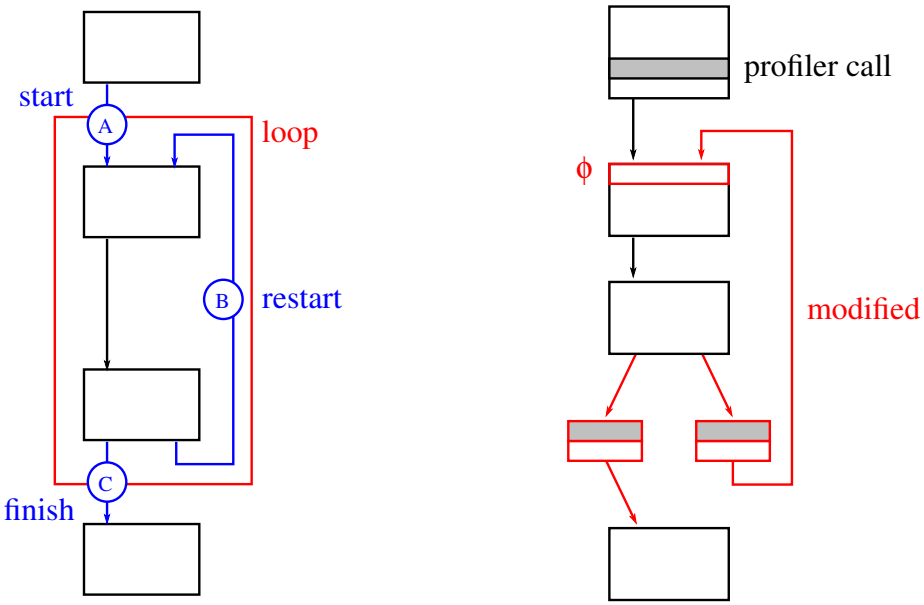
Using these definitions, this section specifies the way loops are instrumented.

A call to **begin_loop()** is inserted immediately before each jump to the entry of a loop. For unconditional branches this is a trivial transformation, but for conditional branches it is more involved. The same is true for calls to **end_loop()** and **next_iteration()**, the difference being that the jump being instrumented is of a different nature. Here is a list to clarify the correspondence:

1. If the jump is a **start** of the loop (see Definition 5; edge (A) in Figure 3.3) - insert a call to **begin_loop**;
2. If the jump is a **restart** of the loop (see Definition 7, Item 2; edge (B) in Figure 3.3) - insert a call to **next_iteration**;
3. If the jump is a **finish** of the loop (see Definition 6; edge (C) in Figure 3.3) - insert a call to **end_loop**.

Note that this covers all possible control flow changes from a loop, including **break**s, **continue**s, and even arbitrary **goto**s. Breaks result in finish edges, continues - in restart edges, and gotos result in any of the three types, depending on the position and target of the jump.

Figure 3.3 illustrates how an example CFG representing a do-while loop is transformed by the algorithm.



(a) Example CFG before loop profiling.

(b) Same CFG after loop profiling.

Figure 3.3: An example control flow graph before and after loop profiling. A profiler call is inserted before the start of the loop. Since the finish and restart result from a conditional branch a new basic block needs to be created for each of the two. In those new basic blocks, a single profiling instruction is inserted and an unconditional jump to the original target. Notice the ϕ node that had to be changed due to the change in the CFG: its second incoming block has changed.

The procedure for adding calls to the three loop profiling functions is described in Algorithm 1. In a nutshell, the algorithm checks whether the branch instruction that results in an edge is conditional or not. If not, then a call to the profiling runtime is simply inserted before the branch instruction. Otherwise, a new basic block is created and injected in place of the edge: the edge is redirected to that new basic block and the basic block is made to point to the original target unconditionally. The profiling call is the only instruction in the new basic block other than the unconditional jump.

ALGORITHM 1: Inserting calls to loop profiling functions. If E is a start then the corresponding function is **begin_loop()**, if it is a restart then the function is **next_iteration()**, and if it is a finish then the function is **end_loop()**.

Input: flow graph F, edge E in F (a start, restart, or finish of a loop)
Result: instruments E in F
create a call C to corresponding loop profiling function (**begin_loop()**, **next_iteration()**, or **end_loop()**);
if *E is the only edge leaving from its source* **then**
    let I be the unconditional branch that results in E;
    let B be the block of F that contains I;
    insert C immediately before I in B;
else
    let I be the conditional branch that results in E;
    let B be the block of F that contains I;
    let T be the target of E;
    create a new block N in F;
    create an unconditional branch J to T;
    insert C, followed by J in N;
    change I so that it points to N instead of T;
    for each ϕ node in T: change references from B to N;
end

Note that a single control flow edge can potentially be a finish of one loop and a start of another (edge (A) in Figure 3.4(a)), or the finish of one loop and the restart of a parent loop (edge (B) in Figure 3.4(a)). For this reason the order of applying algorithm 1 matters, as it defines the order in which calls to the profiling library will be inserted in the code. Thus, start edges should be profiled first, restart edges second, and finish edges last, leading to calls to **end_loop()** prepended to calls to **next_iteration()** prepended to calls to **begin_loop()**.

The reason for this order is twofold. The first key observation is that finish edges always result from conditional jumps. To see why this is the case, consider the possibility that a finish edge results from an unconditional jump. Since the edge is a finish it must be to a block that has no path back to the loop. But then the block that contains the branch instruction has no path back to the loop either, so it cannot be in it and a contradiction is reached.

The second key observation is that an edge cannot be a start of one loop and a restart of another at the same time. This is because **loops** are defined as **maximal regions** (See Section 3.3.2.1) and therefore no two loops can have the same **entry**.

As a result, because there is no need to worry about starts colliding with restarts, the nature of finishes is the defining one - that they come from conditional jumps. Algorithm 1 will insert the profiling call corresponding to the type of the edge **after** the jump. As a result, if calls are first inserted to **begin_loop()**, followed by calls to **next_iteration()**, followed by calls to **end_loop()**, that will guarantee that these will occur in reverse order during execution: **end_loop()** first, followed by **next_iteration()** or **begin_loop()**.

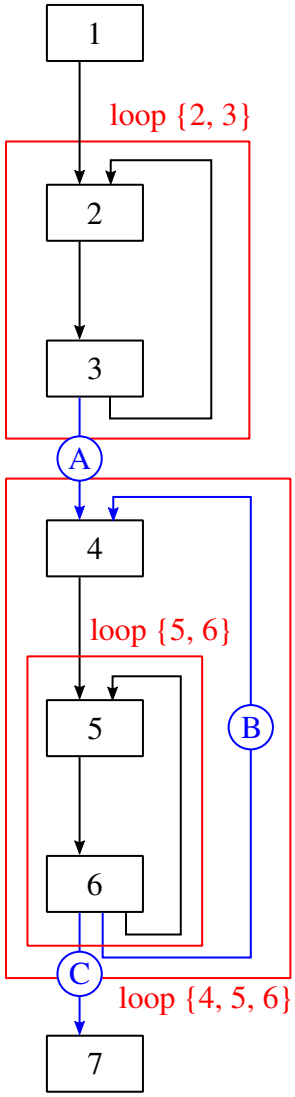
Finally, it is also possible that an edge is the finish of multiple loops at the same time (edge Ⓒ in Figure 3.4(a)). In such cases, multiple calls to **end_loop()** need to be added after the jump instruction that results in the finish edge – as many as the number of loops that need to be finished.

3.3.3 Function Calls

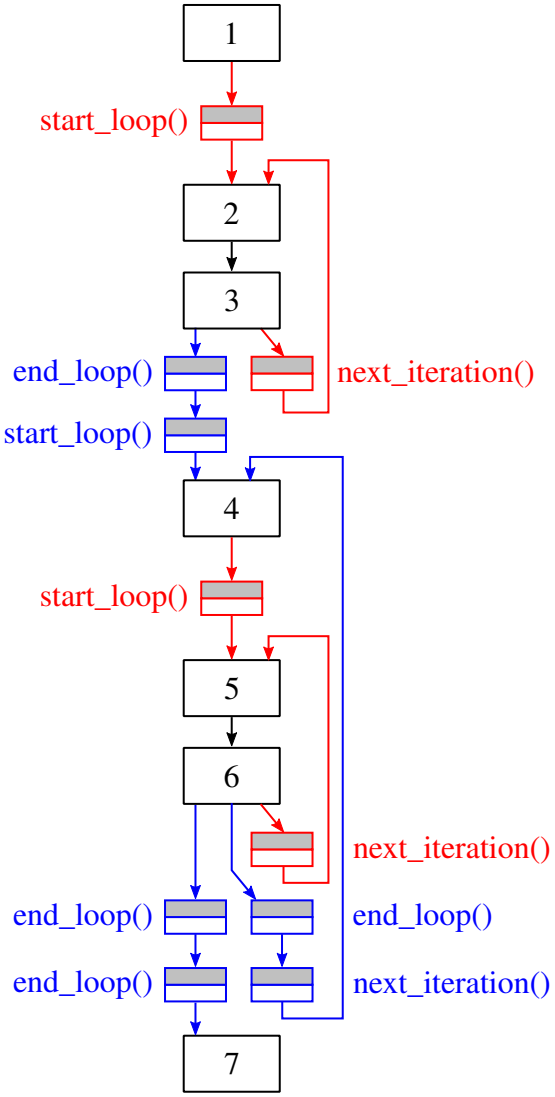
As mentioned earlier, in this chapter, it is assumed that the program under analysis can be statically linked into a single translation unit. Profiling dynamically linked libraries and multiple translation units is left as further work. In order to instrument function calls, a complete static call graph of the program is first computed: a directed multi-graph, the edges of which are labelled with the corresponding call sites. Indirect function calls get in the way of this, and section 3.3.3.2 discusses how to handle them. But first, the problem of recursive functions is discussed.

3.3.3.1 Recursion

One of the major areas for improvement of [45] is the lack of a robust mechanism to handle recursive functions. The approach taken in that master's thesis is the following: 'To prevent from generating a context tree infinitely, the CAMP compiler marks re-



(a) Some CFG edges change control for multiple loops. For example, an edge can be at the same time a finish for one loop and a start for another (edge A), a finish for one and a restart for another (edge B), or a finish for multiple loops (edge C).



(b) The order of instrumentation in such cases matters. If it is reversed it will lead to either inferring non-existent loops and failing to end existent ones (in the former case), or miscounting iterations of parent loops (in the latter case).

Figure 3.4: Examples of CFG edges that change control for multiple loops. There are three cases: finish-start (A), finish-restart (B), and finish-finish (C). Multiple profiling calls need to be added for these edges, whereas single calls are inserted for simple edges.

cursive functions before generating the context tree, and inserts only the first recursive function call site as a leaf loop context node. Here, the compiler considers the recursive function call site node as a loop node, so the CAMP profiler can find its recursion depth by counting iteration numbers.'

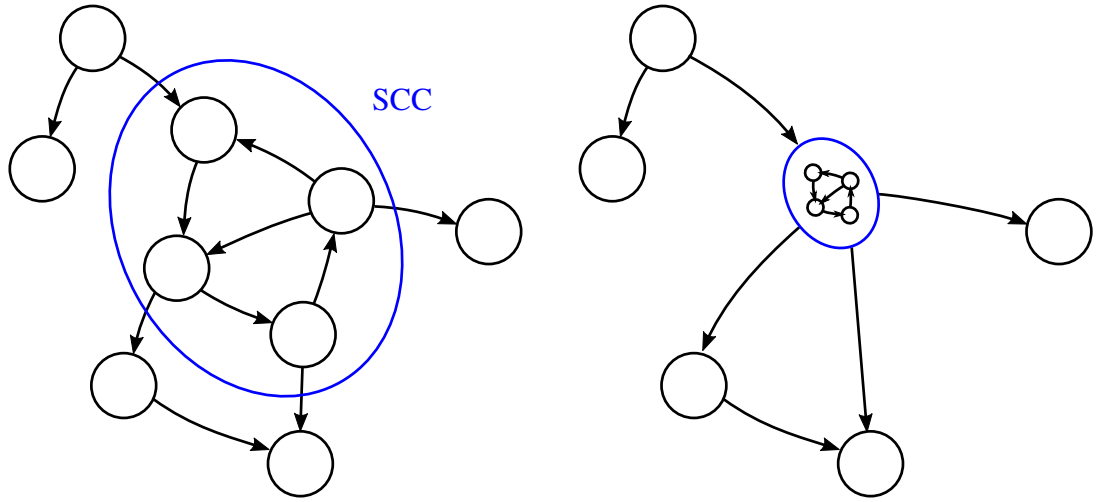
While it is true that some recursive functions can be expressed as loops, this is generally not the case (see [64], [65]). In fact, described in this way, this approach can only be applied to functions directly calling themselves (although the examples in [45] suggest that mutually recursive functions can also be handled). This recursive call should appear immediately before a return instruction that returns the result of the recursive call (also known as a tail-call). Otherwise, the mapping between invocations of the recursive function and the iterations of a loop fails, because 'iterations' of this hypothetical loop will not have a strict order - they would sometimes interrupt their execution and then continue after the execution of a later 'iteration'.

Furthermore, while it seems possible to transform recursive functions to loops in the general case by converting them to tail-recursive functions, [64] and [65] show that this is a non-trivial transformation for humans, let alone for compilers. This is because it involves analysis of the nature of operations performed inside the function, finding identities and arithmetic properties. Thus this chapter does not associate recursive functions with loops.

The last issue with this approach is that it is unclear how calls to non-recursive functions from within a recursive function are handled. If recursive function calls are inserted as leaf nodes, then these calls to non-recursive functions seem to be ignored and thus the result would be inaccurate.

The approach to handling recursion presented here is the following: all strongly connected components in the call graph are detected. If an SCC contains more than one node, or nodes that point to themselves in the original graph, then these nodes form chains of mutually recursive functions. A derivative graph is built – the graph of the strongly connected components of the call graph. For each edge between two nodes of the call graph a corresponding edge between the corresponding SCCs that the nodes belong to is created, unless they belong to the same SCC in which case no edge is created. Figure 3.5 illustrates this approach.

After recursive calls have been handled, paths from the starting node to each node of the SCC graph with unique context IDs are assigned. This is done implicitly, by assigning offsets to the edges of the SCC graph. The SCC graph is necessarily a DAG since cycles have been reduced to single nodes.



(a) An example call graph.

(b) Derived SCC DAG.

Figure 3.5: Handling recursion robustly. The strongly connected component in Figure 3.5(a) is represented by a single node in the derived graph in Figure 3.5(b). Note that the resulting graph does not contain any loops by construction, i.e. it is a directed acyclic graph (DAG).

The recursive Algorithm 2 is performed to assign offsets to the function calls in the SCC graph. Necessary for the recursive accumulation, the algorithm returns the number 'contexts' in the sub-DAG at the node passed as an argument. A 'context' is a unique path from the root of a DAG to one of its nodes and corresponds to a unique call stack at runtime. This number of contexts is represented by the **current_offset** variable which is recursively computed by adding one for each child (because of the new path that leads to it) and the number of contexts in the sub-DAG rooted in that child.

Each child is assigned an offset equal to one plus the current number of sub-contexts that have been taken into account. This results in a unique and compact numbering for each context.

Example 5. Figure 3.7 shows an example call graph with offsets assigned to function calls by applying Algorithm 2. Starting at node A, the edge to node B is assigned offset $1+0=1$. Then offsets are recursively assigned for node B.

Similarly, at node B, the edge to node D is assigned offset 1 and offsets are recursively assigned for node D. There, the edge to node F is assigned offset 1 and the algorithm recurses on node F.

Since node F has no children, it returns 0 as the number of edges in its sub-DAG.

ALGORITHM 2: Assign offsets to all call sites of a call graph. The call graph should have recursive SCCs reduced to a single node to guarantee that it is a DAG. Call on each node of the DAG to obtain offsets of all call sites. The asymptotic complexity of this algorithm is $O(|E|+|V|)$, since each edge and each node are visited exactly once.

Input: Node N of a DAG G

Result: Assigns offsets to all edges in the sub-DAG of G rooted at N. Returns the number of contexts in that sub-DAG.

```
if N has already been visited then
    return cached result
else
    current_offset = 0;
    for each successor S of N do
        offset of edge to S = 1 + current_offset;
        current_offset += 1 + assign offsets for S;
    end
    cache current_offset as the result for N;
    return current_offset
end
```

Stepping back to the call for node D, **current_offset** is updated to $1+0=1$. There are no more children, so 1 is returned as the number of edges in the sub-DAG rooted at D.

Stepping back to node B, **current_offset** is updated to $0+1+1=2$ and offset $1+2=3$ is assigned to the edge to E. Then the algorithm recurses on E, which behaves the same way as the call for D and returns 1. Still at the call for B, **current_offset** is updated to $2+1+1=4$, and because there are no more children, 4 is returned as the number of edges of the sub-DAG.

Stepping back to node A, **current_offset** is updated to $0+1+4=5$. Then offset $1+5=6$ is assigned to the edge to node C and the algorithm recurses on C. This behaves the same way as the call to B and thus assigns the same offsets to the edges to D and E. A notable difference is that the return values for the calls to D and E have been cached and thus are not recomputed again.

Finally, the call for C returns to the call for A and **current_offset** is updated to $5+1+4=10$ which is the total number of (non-zero length) paths in the DAG. These are listed in Table 3.6 with their contexts computed by adding up all the edges in the path.

Path	Context ID
A	0
$A \rightarrow B$	1
$A \rightarrow B \rightarrow D$	2
$A \rightarrow B \rightarrow D \rightarrow F$	3
$A \rightarrow B \rightarrow E$	4
$A \rightarrow B \rightarrow E \rightarrow F$	5
$A \rightarrow C$	6
$A \rightarrow C \rightarrow D$	7
$A \rightarrow C \rightarrow D \rightarrow F$	8
$A \rightarrow C \rightarrow E$	9
$A \rightarrow C \rightarrow E \rightarrow F$	10

Figure 3.6: A list of all paths in the graph of Figure 3.7 and the unique context assigned to each of them by accumulating the offsets assigned to the edges.

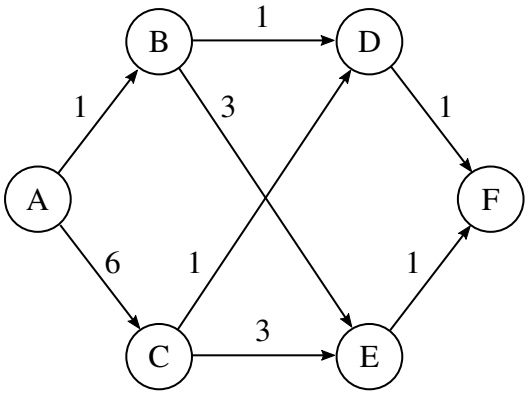


Figure 3.7: An SCC DAG derived from a call graph with associated offsets to its edges: function calls in the program IR.

3.3.3.2 Indirect calls

The other major area for improvement for [45] is the handling of indirect function calls: 'For indirect function calls, the compiler analyses all the possible target candidates and inserts the candidates as children nodes.' A footnote further elaborates: 'Given full access to the source code, the CAMP compiler determines the candidates by matching type signatures of all functions.' While this is reasonable heuristic, it still allows for a potentially exponential growth in the number of possible contexts.

In general, the problem of detecting all possible target candidates for an indirect call cannot be solved statically. This is due to the fact that the result of any arbitrary pointer computation can be used as a function pointer. Using the type signature of the function call is not a scalable solution. There are two approaches to the problem of indirect calls: one is to treat indirect calls as leaves in the call graph and the other is to treat them the same way as direct function calls. While the former results in loss of context precision the latter requires complete recomputation of the call graph every time a new indirect call is performed if it is to be correct.

A hybrid solution is also possible, but for simplicity the first approach was implemented as part of the prototype framework presented here. An example call graph resulting from this approach is depicted in Figure 3.8. See Figure 3.9 for the difference in the offset assignment for an example execution of the call graph in Figure 3.8 versus the offset assignment for the call graph that would result from changing the indirect calls to explicit ones.

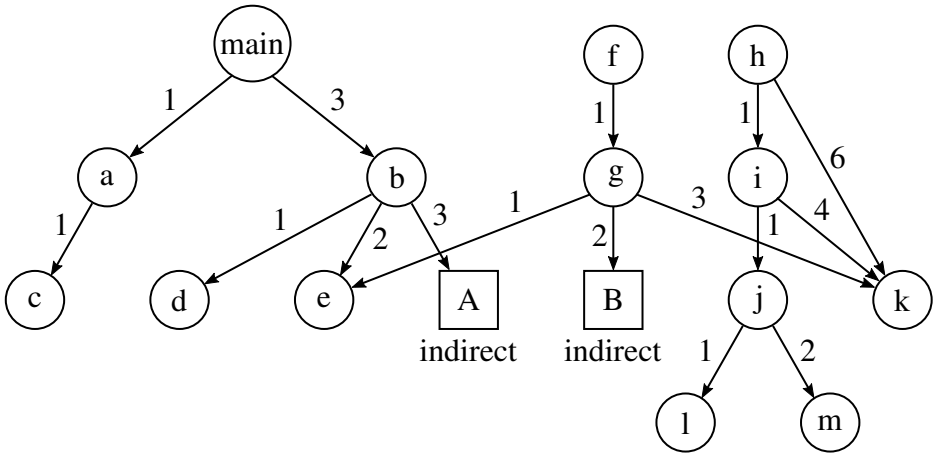
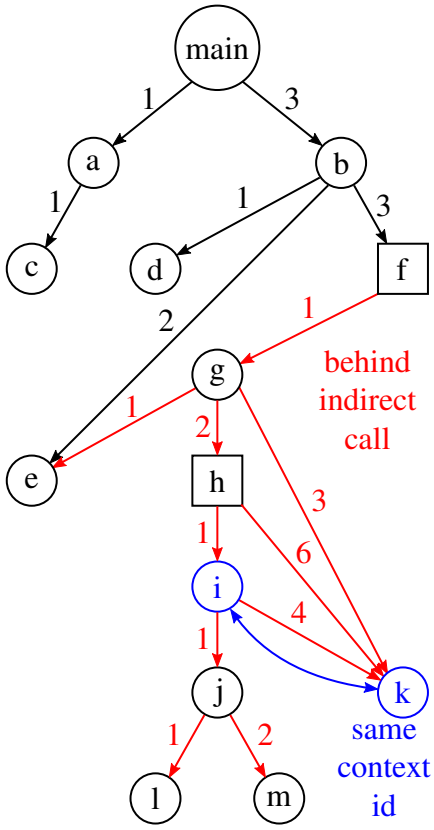
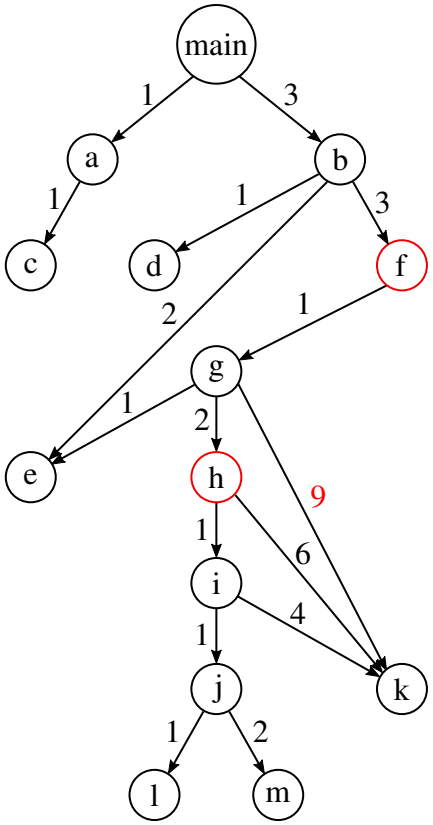


Figure 3.8: Call graph containing indirect calls (represented by squares).



(a) Example execution of the program with call graph shown in Figure 3.8 (A was f and B was h). Note that functions i and k would have the same context ID if indirect calls were not handled as a special case. This would have resulted in incorrect profiling information.



(b) The complete call graph of an equivalent program, that has the indirect function calls replaced with explicit ones. Note the difference in the offset assigned to the call from function g to function k. This is due to the knowledge of the call graph of h (B) in compile time.

Figure 3.9: Comparison of the de facto runtime call graph of the program represented in Figure 3.8 (indirect call A was to function f and indirect call B was to function h) and the static call graph of the same program, but with indirect calls replaced with explicit ones.

3.3.4 Putting it together

After discussing the problems of recursive and indirect calls and the proposed approaches to dealing with them, this chapter presents an integrated methodology of instrumenting function calls. In a nutshell, given a function call the framework checks whether there is a corresponding edge E in the SCC DAG. If not, then the function call is an internal recursive call and it is ignored.

Otherwise, calls to **change_context(+offset)** are inserted and **change_context(-offset)** before and after the instruction that is being profiled. The **offset** variable is the one associated with E by applying Algorithm 2 to the SCC DAG.

Finally, if the function call is indirect then calls to **indirect_push()** and **indirect_pop()** are also inserted before and after it.

Algorithm 3 formalizes this procedure.

3.4 Profiling

3.4.1 Data Structures

There are several runtime variables used to keep track of the state of the profiled program:

1. **context_id** - a non-negative integer representing the current context;
2. **loop_iterators** - a re-sizable array of non-negative integers representing the iteration count of all nested loops currently executing. Note that this includes loops that are nested indirectly via function calls;
3. **indirection_stack_depth** - a non-negative integer keeping track of the number of indirect function calls on the call stack;
4. **top_indirect_context_id** - a non-negative integer keeping track of the context of the first indirect function call on the call stack;
5. **load_iters** - an array of non-negative integers representing the number of initial executions of a load instruction left before sampling is applied. This array is indexed by the IDs of load instructions;
6. **loads** - a map from memory addresses to the set of load instructions that accessed these addresses since the last write to them;

ALGORITHM 3: Inserting calls to function call profiling functions. Calls to recursive functions are ignored if they come from another function in the recursion cycle. All other calls, including calls to recursive functions from non-recursive functions (or recursive functions in a different recursion cycle), are instrumented with two **change_context()** calls to update the context for the function call in question and then restore the context when the call returns. Indirect calls are additionally instrumented with a call to **indirect_push()** and a call to **indirect_pop()** in order to change the behaviour of the memory profiler when there is an indirect call on the call stack.

Input: flow graph F, call instruction I in some block B of F, SCC derivative graph G of the call graph of F

Result: instruments I in F

let E be the edge in G corresponding to I;

if *E does not exist* **then**

    I is an internal recursive call - ignore it;

    return

end

let **offset** be the offset associated with E;

create a call instruction J to **change_context(+offset)**;

create a call instruction K to **change_context(-offset)**;

insert J in B, immediately before I;

insert K in B, immediately after I;

if *I is indirect* **then**

    create a call instruction L to **indirect_push()**;

    create a call instruction M to **indirect_pop()**;

    insert L in B, immediately before I;

    insert M in B, immediately after I;

end

7. **store** - a map from memory addresses to the last store instruction that accessed these addresses;

8. **dependencies** - a set of memory dependencies discovered by the runtime.

A memory access (store or load) is represented by a structure that has the following fields:

1. **access_id** - the identifier of the memory access. This ID is unique within the set of loads and the set of stores, but a load can have the same ID as a store. This does not correspond to any relation between the instructions, but is rather a result of the continuous assignment of identifiers. It allows for a compact storage for the **load_iters** array. The conflicts between loads and stores is not a problem, since they are stored in different data structures (**loads** and **store** respectively);

2. **context_id** - the ID of the context where the memory access occurred. This is either the current **context_id** during the execution of the profiling call, or **top_indirect_context_id** if **indirection_stack_depth** is larger than 0;

3. **loop_iterators** - the **loop_iterators** re-sizable array during the execution of the profiling call. This is independent of the **indirection_stack_depth** variable and can provide additional (although incomplete) locality information whenever there are indirect calls on the call stack.

Finally, a memory dependence is represented by a structure that has the following fields:

1. **type** - a small integer taking values 0, 1, or 2: 0 indicates a RAW dependence, 1 - a WAW dependence, and 2 - a WAR dependence;

2. **from_id** - the ID of the instruction which is depended on, e.g. the store instruction in a RAW dependence or the load instruction in a WAR dependence;

3. **to_id** - the ID of the instruction which depends, e.g. the load instruction in a RAW dependence or the store instruction in a WAR dependence;

4. **from_ctx** - the context ID of the runtime when the instruction which is depended on was executed;

5. **to_ctx** - the context ID of the runtime when the instruction which depends was executed;

6. **from_loop_iterators** - a copy of the re-sizable array of loop iterators at the point when the instruction which is depended on was executed;

7. **to_loop_iterators** - a copy of the re-sizable array of loop iterators at the point when the instruction which depends was executed.

3.4.2 Routines

When the program is started **load_iters** is allocated to be as large as the total amount of load instructions. Then each of its elements is initialized to be **INIT_ITERATIONS** - the constant indicating how many initial executions of each load instruction should be taken into account before sampling starts. Also during the startup of the program, the random number generator is initialized to be used for sampling.

When a store instruction is encountered (**profile_store** is called) the runtime checks if there are any loads to the same address in the **loads** map and creates WAR dependencies between them and the store instruction just encountered. If there is a store instruction to the same address in the **store** map, a WAW dependence is created. After that the new store instruction is stored in the **store** map for the current address and the load instructions in the **loads** map for the current address are erased, if there were any. Any dependencies created in this function call are stored in **dependencies**.

When a load instruction is encountered (**profile_load** is called) the runtime checks the counter stored for that instruction in **loop_iterators**. If it is positive it is decreased and the instruction is profiled. If it is zero, then using the sampling frequency specified by **SAMPLE_DIVISOR** it is randomly decided whether to profile the instruction or not. If the instruction is to be profiled it is added to the set that is associated to the address being loaded inside the **loads** map. Furthermore, if there is a store associated to the same address in the **store** map, a RAW dependence is created and stored in **dependencies**.

When a loop start is encountered (**begin_loop** is called) the runtime adds a new element to the **loop_iterators** re-sizable array and when a loop finish is encountered (**end_loop** is called) the runtime removes the last element of the **loop_iterators** re-sizable array. When a loop restart is encountered (**next_iteration** is called) the runtime increments the last element of the **loop_iterators** re-sizable array.

When a call instruction is encountered or returns (**change_offset** is called) the runtime accumulates the given offset to the **context_id** variable. When an indirect function call occurs (**indirect_push** is encountered), **indirection_stack_depth** is compared to

zero. If it equals, then **context_id** is stored in **top_indirect_context_id**, otherwise - it is not. Then **indirection_stack_depth** is incremented.

Finally, when an indirect function call returns (**indirect_pop** is encountered), **indirection_stack_depth** is decremented. If it then equals zero, **top_indirect_context_id** is set to zero to indicate that there is no indirect call on the call stack.

3.5 Evaluation

The instrumenting and profiling framework was implemented on top of the LLVM compiler toolchain (version 3.9) [56]. For the C and C++ benchmarks the established clang frontend was used, while for the Fortran benchmarks the flang frontend, recently open-sourced by NVIDIA [2], was used. The evaluation of the profiling framework presented here consists of building data access profiles for all of the SPEC CPU2006 benchmarks and reporting the runtime slowdown (Figure 3.10) and the memory overhead of the profiler (Figure 3.11). The hardware system used for the experiments has four AMD Opteron 6376 CPUs (64 logical cores in total) and has 1TB of RAM available. For inputs for the programs the `test` input set provided with the benchmarks were used.

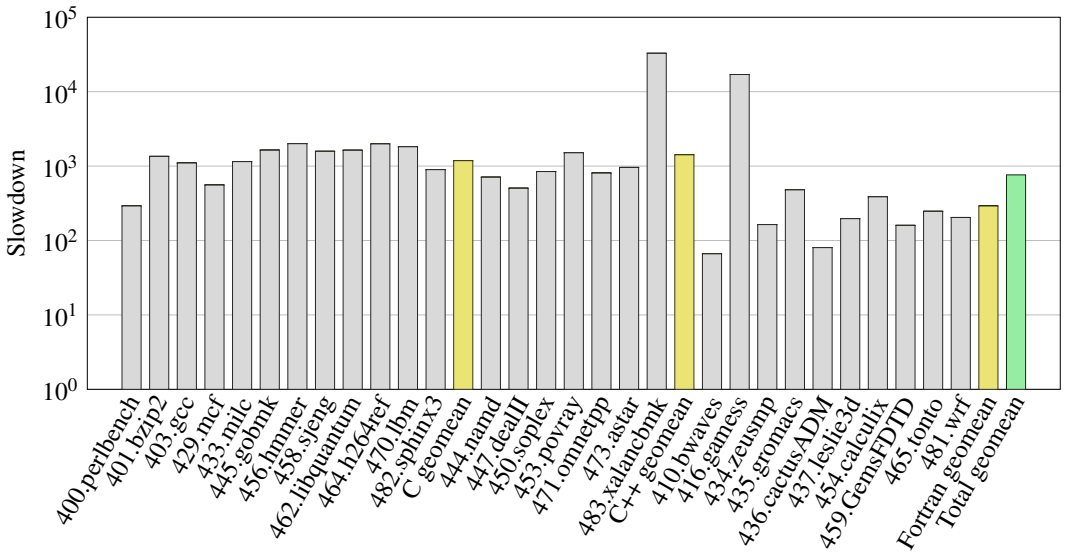


Figure 3.10: Performance slowdown of the SPEC CPU2006 benchmarks due to profiling. The geometric mean of the slowdown for the C benchmarks is $1180\times$, for the C++ benchmarks is $1420\times$, and for the Fortran benchmarks is $290\times$. The geometric mean across all benchmarks is $760\times$.

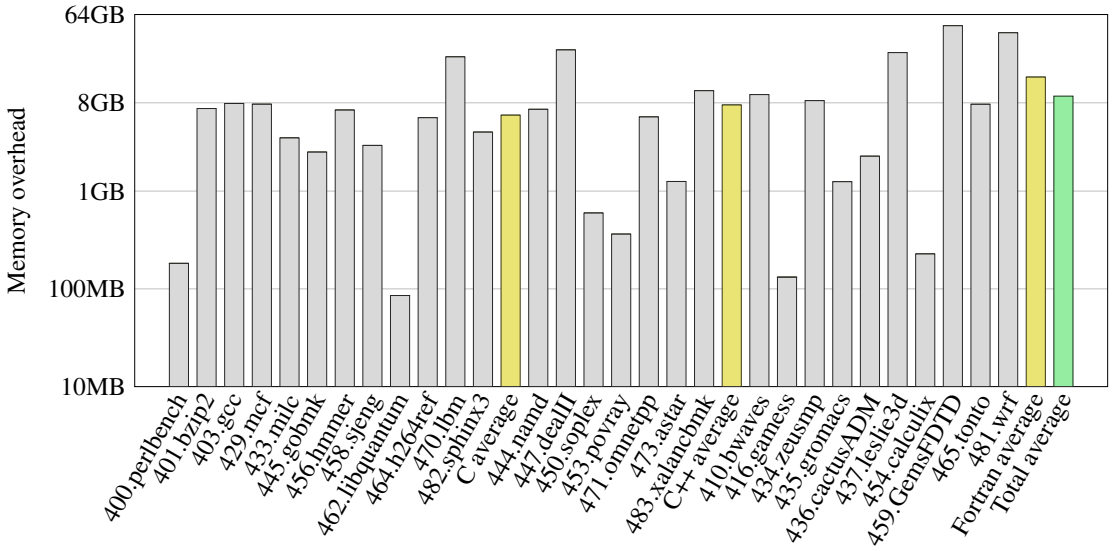


Figure 3.11: Memory overhead of data dependency profiling. The average overhead for the C benchmarks is 6.0GB, for the C++ benchmarks is 7.6GB, and for the Fortran benchmarks is 14.7GB. The average across all benchmarks is 9.4GB.

The application of the technique presented in this chapter results in an average slowdown of 760× and an average memory overhead of 9.4GB. This is worse than state-of-the-art profiling frameworks, some of which claim slowdown as low as 20×. However, as discussed in Section 3.6, none of these faster profiling frameworks has been reported to cope with the whole SPEC CPU2006 benchmark suite. The longest it takes to profile a benchmark is 34 hours, which is still acceptable, given that trying to use a faster technique might yield no results at all. The goal of maximum applicability of the technique has been achieved, as no other profiling framework that the author is aware of reports results for all of the SPEC CPU2006 benchmarks. The framework also achieves instruction-level accuracy, and the coverage of the generated profile is limited by the default input sets provided with the SPEC benchmarks.

3.5.1 Runtime

Figure 3.10 shows that there are two benchmarks for which the runtime slowdown is an order of magnitude larger than for the rest. The reason for this discrepancy is not completely clear, but in both cases the runtime of the unprofiled benchmark is close to the lower limit of precision of the measurement tool (the Linux bash command `time`): in the order of hundreds of milliseconds. As a result, it is possible that the huge

slowdown is more due to the low precision of the measurement tool for this lower limit, rather than any structural feature of the benchmarks in question. These benchmarks are 416.gamess with a slowdown of $17,000\times$ and 483.xalancbmk with a slowdown of $32,750\times$. At the same time, from Figure 3.12 it can be seen that the former takes only 30 minutes to execute: a lot less than the average of seven hours, while the latter takes eight hours: close to that average. For this reason the high slowdown is not considered to be a critical problem.

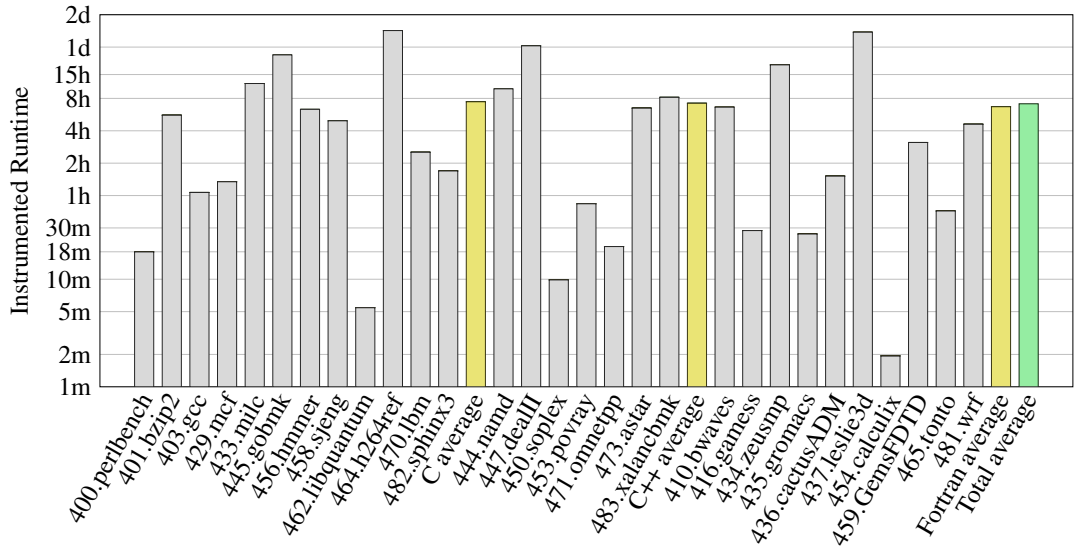


Figure 3.12: Runtimes of the profiled versions of benchmarks. The average runtime for the C benchmarks is 7.5 hours, for the C++ benchmarks is 7.2 hours, and for the Fortran benchmarks is 6.7 hours. The average across all benchmarks is 7.1 hours.

3.5.2 Memory Overhead

Figure 3.11 shows that there is a wide range of memory overhead for the different benchmarks. While it can be as high as 49.2GB and 41.6GB for benchmarks like 459.GemsFDTD and 481.wrf, it can also be as low as 85MB, 131MB, or 181MB for benchmarks like 462.libquantum, 416.gamess, and 400.perbench. The amount of memory used by profiling dominates the amount of memory required by the programs themselves. On average it is 98.6% of the total memory consumed, and ranges from 94.2% to 99.9%. This means that there are some program-input combinations which the framework presented here is better suited for in the case where there is a limited amount of memory available, but this chapter shows that the proposed framework can profile all of the SPEC CPU2006 benchmarks on a computer that has access to 64GB

of main memory.

3.5.3 Choice of Input

When using the `test` input set, the profiler presented in this chapter achieved the coverage reported in Figure 3.13. The figure shows that the percentage of the total number of loops in all benchmarks that is exercised during profiling is only 19. Intuitively it may seem that using the larger `ref` input set instead might give better coverage results. However, after running eight of the benchmarks with the `ref` input set, it turned out that there is little improvement in coverage that can be achieved in this way. For four benchmarks the loop coverage did not change at all, for two others the increase was negligible (2 and 3 additional loops out of 297 and 329, respectively), and, surprisingly, for one of the benchmarks two loops less were covered. There was only one benchmark, 445.gobmk, which saw a significant increase in the coverage, raising from 23% to 73% of the total loops in the benchmark. Time for profiling, however, increased 185× on a geometric average over the eight benchmarks that were measured.

A workaround for the problem of limited coverage is to devise a test harness dedicated for the task of profiling. While it should still be representative of real-use values, code regions of interest could be isolated and profiled independently, in order to minimise the runtime and memory overhead of profiling. Techniques from software testing could be employed, for example unit testing, in order to maximise the code coverage, while minimising the execution time. The design and collection of such input sets, however, is outside the scope of this thesis.

3.6 Related Work

Dynamic data access profiling has been employed for measuring and detecting parallelism for several decades, with important early work developed by [52] and [55]. In [52], Kumar et al. aim to measure the amount of parallelism available in a set of computation-intensive scientific applications. They introduce the notion of shadow memory that keeps information about memory accesses, and control variables to compute statistics of the directions taken at control branches. In [55], Larus et al. present `pp`: a system that first computes a trace of a program that includes memory access information and then uses that trace to simulate the parallel execution of the program and predict its performance. Chen et al. are the first to investigate the problem of

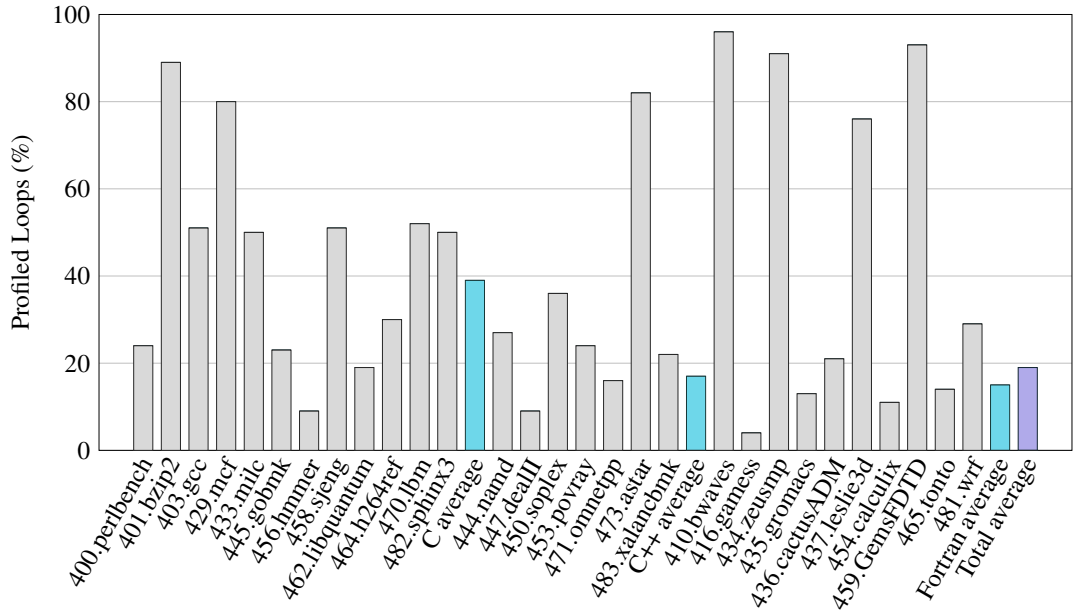


Figure 3.13: Percentages of loops covered by the SPEC CPU2006 `test` data set compared to all loops contained in the benchmarks. The average coverage of C benchmarks is 39%, of C++ benchmarks is 17%, and of Fortran benchmarks is 15%. The total average coverage is 19%.

software-only data dependence profiling, in [23]. The authors introduce the fundamental hash-based memory access recording technique and frequency sampling for runtime reduction that are used in this chapter too. However, since the target application is speculative optimisations, their framework focuses on solving relevant problems, e.g. computing dependence probabilities, and thus is not as general.

The framework described in this chapter is inspired by the CAMP profiler [45] and as such shares many its features. CAMP handles recursive functions as loops, which this chapter shows is not general enough to describe all types of recursion. Instead, groups of mutually recursive functions (indicated as strongly connected components of the call graph) are treated here as a single context. While this adds some inaccuracy to the location of instructions in these recursive groups, it solves the problem of the explosion of the amount of contexts as a result of generating a new one for every recursive call. The other major difference from CAMP is the way the framework presented here handles indirect function calls. CAMP generates contexts for all possible functions that can be called at every indirect call site using the type signatures of functions. While this heuristic somewhat reduces the number of possibilities there is still a potentially exponential growth in the number of unique contexts when there are many functions with the same signature in the program. Instead, this framework notifies the

runtime when an indirect call site is encountered and the context is not updated until that call returns. Similarly to the approach for recursive functions presented here, this results in loss of accuracy, but an increase in robustness. This is demonstrated by the fact that evaluation of the technique involves all 29 of the SPEC CPU2006 benchmarks compared to the six benchmarks from the CINT2006 subset that CAMP is evaluated on.

While this chapter develops a technique with the main goal of wide applicability and then the goal of accuracy, other work focuses on minimizing the execution time of profiling. The SD^3 system, presented in [46], addresses runtime and memory overhead for data dependence profiling in a scalable way. Kim et al. employ a compression algorithm to deal with the rapidly growing memory needs of deep loop nests, and they also use a hybrid pipeline and data-level parallelisation to address performance concerns. However, SD^3 does not take into account different call sites when recording dependencies and thus does not achieve the accuracy that is aimed for in this chapter. Also, it is not clear how the SD^3 system handles recursive calls if at all. The techniques from SD^3 can be leveraged to optimize the framework presented here in a similar manner, but this is outside the scope of this thesis.

Parwiz, presented in [44], uses dynamic binary instrumentation to insert calls to a profiling runtime directly in the compiled program. Then it collects dynamic dependence information and combines that with static dependence analysis in order to suggest parallelisation opportunities to the user programmer. It uses coalescing of consecutive memory accesses in order to represent data structures on the heap as single objects. Furthermore, program loops which can be categorised as static control loops (i.e. the control does not depend on the computation of the loop) are parametrised: the memory accesses they make are described by storing the loop parameters. These two techniques allow Parwiz to reduce its memory and runtime overheads.

Another binary level profiler is developed in [88]. Sato et al. represent executed programs as context trees that contain both function calls and loop invocations, similar to the approach taken in this chapter. The nodes of this tree are further connected by the dependencies discovered by the runtime, essentially producing a dependence overlay of the tree. The authors present a way to use this representation to identify different types of parallelism in the profiled program. While the focus of that work is on a low runtime overhead and the accuracy of the profiler is good enough for the application for which it is used, it is a lot coarser than the accuracy that the framework presented in this chapter achieves.

In [95], Vanka et al. try to find a better trade-off between speed and accuracy of data dependence profiling for speculative optimization than previous works. The proposed approach is to group instructions into sets and track dependencies between these sets only. The sets are computed depending on the needs of the optimisation which will use the profiling information. This leads to improved runtime and good-enough accuracy for the optimisation of choice. However, being tailor made for a specific optimisation, the profiling information is not as general as the one produced by the technique presented here.

In [100], Yu et al. also focus on reducing the memory and performance overhead of data dependence profiling. The proposed approach involves using type consistency and alias analysis to group memory locations into alias clusters, using a partial dependence graph without enumerating all dependencies, and partitioning the profiling into complementary slices. This allows them to run profiling in parallel and reduce the memory requirement of each slice. The authors prove that using a partial dependence graph is sufficient for the loop-level reordering transformations that they are interested in enabling, but it is not general enough for the applications the approach presented here aims to enable.

3.7 Summary and Conclusions

When it comes to data accesses, static analysis can derive a limited amount of information. In order to extend this information, this chapter has presented a dynamic context aware data access profiling framework. It extends previous work [45] by implementing robust mechanisms for handling recursive and indirect function calls. Furthermore, formal definitions for loop related terms are presented and the technical details of the framework are explicitly described in order to facilitate future improvements.

The framework presented in this chapter achieves its main goal of being complete enough to be able to profile the whole of the SPEC CPU2006 benchmark suite, unlike more complex techniques. At the same time, the precision is of the level required for augmenting the iterator recognition analysis presented in the next chapter: dependencies are recorded on the instruction level and relative to a context including the call-stack and inter-procedural loop nesting. The framework is not as fast as other methods, but still manages to profile each SPEC CPU2006 benchmark in an acceptable time. If runtime efficiency and memory overhead are critical for an application, techniques from more complex profilers can be employed to extend the profiler, in

particular parallelising by pipelining the tracer and analyser steps and running multiple trace analyses in parallel, and by compressing the collected information before analysing it as proposed by [46].

Chapter 4

Generalized Loop Iterator Recognition

Iterators prescribe the traversal of data structures and determine loop termination, and many loop transformations require exact knowledge of iterators for their analysis and manipulation. While recognition of iterators is a straight-forward task for loops with affine loop indices, the situation is different for loops with more complex iterators, e.g. iterating over dynamic data structures or involving control flow dependent computation to determine the next loop element. This chapter proposes a compiler analysis for recognizing and separating out loop iterator code from instructions representing the loop "payload". A static analysis is initially developed for this task, which is then enhanced by incorporating profiling information to support speculative code optimizations. The new analysis has been prototyped in the LLVM framework and demonstrate its capabilities using the SPEC CPU2006 benchmarks. The approach is applicable to all loops and this chapter shows that it can recognize explicit iterators in, on average, 88.1% of over 75,000 loops using static analysis alone, and up to 94.9% using additional profiling information. Existing techniques perform substantially worse, especially for C and C++ applications, and cover only 35–44% of the loops. The analysis presented here enables advanced loop optimizations such as decoupled software pipelining, commutativity analysis and source code rejuvenation for real-world applications, which escape analysis and transformation if loop iterators are not recognized accurately.

4.1 Introduction

Advanced compiler optimization [91] is largely concerned with the optimization of loops as the largest proportion of time executing a program is spent in loops, where small per-iteration improvements multiply to greater overall effect. In fact, mathemat-

ical frameworks – such as the polyhedral model [15] – underpinning loop optimization and parallelisation have been specifically designed to capture loop behaviour for analysis and transformation. Central to loop analysis is knowledge of loop iterators, which determine the iteration ranges of loops, are updated in every iteration and prescribe how data structures are traversed. Whereas recognition of loop iterators for Fortran-style `do`-loops is trivial and can be accomplished using syntactic pattern matching, most compilers employ some variation of induction variable recognition at the intermediate representation level to capture a wider range of iterators resulting from a multitude of idioms and programming styles.

While advanced loop optimizations have been developed in the academic literature their deployment in commercial or open-source compilers such as LLVM has been hampered by the fact that real-world source code often defeats existing compiler analyses. The generalized iterator recognition analysis developed in this chapter represents a critical step towards enabling advanced transformations to be applied to a wider range of codes. To motivate the technique developed in this chapter three loop analyses and transformations are considered, which have in common that each of them relies on the separation of code constituting the loop iterator from the rest of the code, forming the per-iteration "payload" of the loop.

Use Case 1: *Decoupled software pipelining (DSWP)* [84], a parallelisation technique for loops with a loop-carried dependence through a pointer-chasing load. While the initial implementation of this transformation in [84] relies on ad-hoc syntactic pattern matching for iterator recognition, later versions [93, 40] use an enhanced analysis for splitting "recursive data structure (RDS) traversal loops" into separate loops for traversal and per-element processing, respectively. Still, DSWP iterator recognition is currently limited to pointer-chasing loops without complex inner control flow or function calls.

Use Case 2: *Commutativity analysis* [85] aims at automatically parallelizing computations that manipulate dynamic, pointer based data structures. In principle, commutativity analysis enables parallelisation in the presence of data dependencies, but its underlying algorithm is restricted to just two forms of `for`-loops with constant bounds and increments. In fact, the combined restrictions'' of the formalism and prototype implementation prohibit application of commutativity analysis to e.g. SPEC CPU2006, where every single benchmark defeats loop commutativity analysis. Among the reasons is – apart from practical issues relating to the prototype implementation – that the formalism developed in [85] cannot handle loops in a general way due to its inability

to identify generalized iterators.

Use Case 3: *Source code rejuvenation* [77] leverages enhanced program language and library facilities by finding and replacing coding patterns that can be expressed through higher-level software abstractions. This includes, among other transformations, the identification of user-defined container data structures, e.g. singly-linked lists, their traversals and replacement with equivalent STL or Boost containers and iterator methods. It is clear that in order to replace user-defined iterator code over complex, possibly recursively defined and dynamically created data structures these iterators need to be identified and separated from the operations performed on the elements of the data structure.

4.1.1 Motivating Examples and Use Cases

Consider the affine loop in Figure 4.1, where the iterator `i` can be trivially recognized using syntactic pattern matching[1]. Optimizing and parallelizing compilers typically rely on the recognition of basic loop iterators to enable further analyses and transformations [62]. Induction variables induced by the surrounding loop (not present in this example) can be identified and substituted by closed form expressions using techniques developed in e.g. [81, 80].

```
1 int array[100];
2 ...
3 for(i = 0; i < 100; i++) {
4   array[i]++;
5 }
```

Figure 4.1: A simple affine loop, where the highlighted loop iterator is trivially recognized using syntactic pattern matching. Further induction variables depending on this iterator `i` may be recognized using techniques developed in e.g. [81, 80, 62].

Now consider the example in Figure 4.2. This loop traverses a recursive data structure – a singly-linked list – and applies a local update to each element of the list. It is clear that this loop does not have a natural iterator in the sense of an integer value loop index incremented by a fixed amount in every iteration [81], but instead a pointer is

[1] Please note that examples in this chapter are shown using C/C++ source code for ease of illustration, whereas the techniques and their implementations operate on internal representations such as LLVM IR and are language agnostic (with exception of source code rejuvenation in Figure 4.3).

updated and checked. Such pointer chasing iterators can be recognized using ad-hoc pattern matching [84] or using an algorithm developed as part of DSWP+ [93, 40]. Following successful recognition of the loop iterator, highlighted in the example, the loop can be parallelized using the decoupled software pipelining approach.

```
1 my_list ptr;
2 ...
3 while(ptr) {
4   ptr->val++;
5   ptr = ptr->next;
6 }
```

Figure 4.2: A traversal of a recursive data structure loop with highlighted iterator code. RDS loop iterators can be recognized using ad-hoc pattern matching [84] or using a partitioning algorithm developed as part of DSWP+ [93, 40].

An alternative use of iterator recognition is in source code rejuvenation [77], where the loop from Figure 4.2 may be converted to the form shown in Figure 4.3, where the user-defined singly-linked list and its traversal have been replaced with an STL `list` container and its hidden iterator methods, another common programming language abstraction of iterators offered by the C++ programming language. Generalized iterator recognition enables this kind of loop transformation aimed at raising code abstraction.

```
1 std::list<int> list;
2 ...
3 for(int &val : list) {
4   val++;
5 }
```

Figure 4.3: The loop from Figure 4.2 after source code rejuvenation, where the traversal of the user-defined linked list has been replaced with an equivalent STL `list` container and abstracted iterator methods. Iterator recognition enables further analyses driving this kind of loop transformation.

The third example in Figure 4.4 is a code excerpt showing breadth-first graph search implemented in the C++ programming language. Conceptually, the loop spanning lines 15–30 of this example is similar to the loop in the pointer-chasing loop in

Figure 4.2, but now the code makes use of STL's `list` container to store lists of adjacent nodes (`adj`) and a queue of nodes (`queue`) needed for BFS traversal. Within the loop, methods for checking and updating the iterator are invoked. Additionally, the loop contains an inner loop in lines 24–29 with its own iterator and further conditional inner control flow, adding to the complexity of the code contributing to the traversal of the graph data structure. Conceptually, we can say that a BFS graph iterator involves a queue and operations on it as well as an additional loop to enqueue newly discovered nodes, which matches the expectations for this algorithm.

In this example, we can separate out the output statements in line 18, which do not contribute to the loop traversal and its termination, and therefore consider to form the loop "payload". RDS loop recognition [84, 93, 40] is defeated by this loop comprising method invocations and complex control flow, whereas the technique presented in this chapter successfully identifies the code forming the (highlighted) iterator of the outer `while`-loop. Accurate recognition of the iterator in this example can be used to enable further analysis, e.g. to drive DSWP-style parallelisation or loop commutativity analysis [**?**].

The examples in this section exemplify different notions of iterators in use today, which require different analysis for their recognition. They also demonstrate that current techniques for iterator recognition are limited in their ability to process complex iterators, which may involve additional control flow or function calls. As a result, the success of advanced loop transformations which require separation of loop iterator code from the payload is hampered. What is needed is a technique capable of processing real-world codes employing a broad range of different styles of loop iterators, which may make use of user-defined data structures, STL or Boost containers and iterators, and complex control flow alike.

4.1.2 Contributions

This chapter initially proposes a new definition of generalized loop iterators, which subsumes existing notions of iterators including Fortran-style DO-loop iterators, affine loop iterators, pointer-chasing iterators and object-oriented iterators, e.g. `it.begin()` or `it.next()`. A novel iterator recognition algorithm is then developed, which operates at the intermediate representation level and which can be used to separate iterator code from the "payload" for any loop. This includes iterators which make use of STL or Boost constructs, or involve nested, conditional control flow or function calls. This

```
1  void Graph::BFS(int s)
2  {
3    // Mark all the vertices as not visited
4    bool *visited = new bool[V];
5    for(int i = 0; i < V; i++)
6      visited[i] = false;
7
8    // Create a queue for BFS
9    list<int> queue;
10
11   // Mark the current node as visited and enqueue it
12   visited[s] = true;
13   queue.push_back(s);
14
15   while(!queue.empty()) {
16     // De-queue a vertex from queue and print it
17     s = queue.front();
18     cout << s << " ";
19     queue.pop_front();
20
21     // Get all adjacent vertices of the de-queued
22     // vertex s. If a adjacent has not been visited,
23     // then mark it visited and enqueue it
24     for(auto i : adj[s]) {
25       if(!visited[i]) {
26         visited[i] = true;
27         queue.push_back(i);
28       }
29     }
30   }
31 }
```

Figure 4.4: Breadth-first search of a graph implemented in C++ and using STL's `list` container to store lists of adjacent nodes (`adj`) and a queue of nodes (`queue`) needed for BFS traversal. The iterator of the `while`-loop spanning lines 15-30 comprises complex control flow and method invocations, and can be separated from the loop "payload" in line 18, which prints a vertex id.

chapter develops two versions of the iterator recognition algorithm: (a) based on static analysis only, which is fast, but conservative, and (b) using additional profiling information to enable more aggressive speculative optimizations. The LLVM prototype implementation is evaluated against the SPEC CPU2006 benchmark suite and its ability to successfully recognize more loop iterators than any other technique is demonstrated.

4.1.3 Overview

The remainder of this chapter is structured as follows. Section 4.2 provides the background on existing techniques for iterator recognition, in particular, induction variable recognition and RDS loop partitioning. This is followed by the presentation of a new technique in Section 4.3, incorporating both static and profiling information, and capable of driving more aggressive speculative optimizations. Relevant details of the LLVM prototype implementation are also provided. Section 4.4 evaluates the new technique and demonstrates performance results. This is followed by a discussion of related work in Section 4.5 and further analyses in Section 4.6 before Section 4.7 summarizes and concludes.

4.2 Background

This section briefly revisits existing notions of iterators, namely affine loop iterators, induction variables, object-oriented iterators and iterators of recursive data structure (RDS) loops. Loops which do not have any iterators at all or where iterators are inherently and inseparably tied to the loop computation are also observed.

4.2.1 Affine Iterators

In compiler theory iterators are typically associated with structured `for`-loops, where a normalized loop iterates over a sequence of consecutive integer numbers between affine lower and upper bounds. The iteration space of such a loop or loop nest is an ordered set of loop iterations, in which each iteration is represented by a point $(i_1,\ldots,i_n)$ for a loop nest of depth n. Loop iterators are then the space represented by a column vector $\mathbf{I} = [i_1,\ldots,i_n]^T$, and loop ranges form a system of inequalities $LB_n(i_1,\ldots,i_{n-1})^T \leq i_n \leq UB_n(i_1,\ldots,i_{n-1})^T$, where LB_k and UB_k are lower and upper loops bounds, respectively, at nesting level k.

4.2.2 Induction Variable Recognition

Informally, an induction variable is a variable whose value on each loop iteration is a linear function of the iteration index (=iterator). A *basic induction variable* is a variable X whose only assignments within the loop are of the form $X \leftarrow X + C$ or $X \leftarrow X - C$, where C is a constant or a loop-invariant variable. More generally, an *induction variable* is recursively defined as either a basic induction variable or a linear function of some induction variable.

Induction variable recognition uses a reaching definitions analysis, finding all definitions of X and Y in $X \leftarrow Y \oplus Z$, which reach the beginning of the loop. Basic induction variables can then be found by a simple scan of the loop. To find the remaining induction variables, we find variables W such that $W \leftarrow A \times X + B$, where A and B are constants or loop invariants, and X is an induction variable. These can be found by iterating through the loop until no more induction variables are found.

Approaches to induction variable recognition in many modern compilers, including LLVM, are based on scalar evolution, e.g. [79]. The approach to iterator recognition that is presented in this chapter is compared against existing techniques and, in particular, the LLVM associated Polly tool [35], which provides convenient access to this functionality.

4.2.3 RDS/DSWP Loop Partitioning

Parallelisation of *recursive data structure (RDS)* loops is the main concern of DSWP [84]. In order to parallelize loops DSWP must identify which pieces of code are responsible for the traversal of the recursive data structure (= iterator recognition). Since a data structure is recursive if elements in the data structure point to other instances of the data structure, either directly or indirectly, RDS loops can be identified by searching for this pattern in the code. Specifically, DSWP searches for load instructions that are data dependent on previous instances of the same instruction. These *induction pointer loads (IPL)* form the kernel of the traversal slice [73]. IPLs can be identified using augmented techniques for identifying induction variables [31, 73]. While RDS loop detection involves iterator recognition for a class of loops comprising pointer based memory references it is based on matching a specific code pattern and does not generalize to a broader class of loops and iterators.

RDS loop recognition has been further developed in [75, 93, 40] as part of DSWP+. Through manual transformation and parallelisation it tries to exploit parallelism on

multicore hardware. The relevant part here is that DSWP+ comprises an algorithm for systematic separation of dependence cycles, some of which may be involved in loop iterators. For this DSWP+ builds the *program dependence graph (PDG) [28]* of the loop. It contains all data (both register and memory) and control dependencies, both intra- and inter-iteration. Then DSWP+ finds the loop recurrences, i.e. instructions participating in dependence cycles. DSWP+ groups dependence cycles into *strongly-connected components (SCCs)* that form an acyclic graph. These SCCs form the minimum scheduling units so that there are no cross-thread cyclic dependencies. The simple algorithm from [75] is used as a starting point for the novel iterator recognition technique presented here and enhanced with support for arbitrarily complex iterators such as those often encountered in STL-enhanced C++ code. The profiling framework from Chapter 3 is also used in order to enhance accuracy for otherwise hard-to-analyse applications.

4.2.4 Object-oriented Iterators

On a programming language level iterators often have a broader meaning [47, 17] and include constructs such as C++ STL iterators, which often escape the stricter iterator notion developed in compiler theory. Specifically, in C++ an iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (`++`) and de-reference (`*`) operators).

This chapter generalizes these existing notions of iterators and introduces a uniform definition subsuming all of the existing, previous definitions of iterators. A technique is developed, which is generally applicable and covers all affine and non-affine, regular and irregular as well as object-oriented iterators in a single, unified framework.

4.2.5 Iteratorless and Inseparable Loops

Not all loops have iterators, which naturally advance the position in an iteration space or data structure. For example, consider the spin-lock loop in Figure 4.5, which spins until a flag is set by e.g. an interrupt handler or another thread. Another example of an iteratorless loops is an infinite event loop, possibly as part of a GUI.

In some cases iterators can not be separated from a distinct loop “payload”, i.e. code which does not contribute to the advancement of the loop iterator. An example of such an inseparable loop is shown in Figure 4.6. While inseparable, this loop may still

```
1 while (!flag) {}
```

Figure 4.5: An example of an iteratorless loop, where loop termination is determined by setting a flag externally, e.g. in an asynchronous interrupt handler or a different execution thread.

be speculatively parallelisable. For example, the technique described in [93] breaks the loop exit control dependencies originating from loop condition in line 3 to facilitate speculative DSWP.

```
1 cost = 0;
2 node = list->head;
3 while(cost < T && node) {
4   ncost = doit(node);
5   cost += ncost;
6   node = node->next;
7 }
```

Figure 4.6: An example of an inseparable loop from [93], where the loop iterator and the "payload" cannot be separated. Inseparable loops may still be parallelisable using speculative DSWP.

4.3 Methodology

This section presents a methodology for iterator recognition. Initially, an overview is provided and both an intuitive and a formal definition of the concept of iterators is given. This is followed by a static analysis for iterator recognition, which – as a useful addition for speculative loop transformation – is subsequently complemented with profiling information to significantly enhance its capability to separate iterator code. Finally, this section shares insights from the LLVM prototype implementation.

4.3.1 Definitions

Intuitively, a loop iterator is a variable (or a set of variables), which is updated in every iteration of a loop and which is involved in controlling loop exits, e.g. as part

of a conditional expression. This intuitive understanding is captured in the following definition:

Definition 4.3.1. *Generalized Loop Iterator.* A generalized loop iterator is a minimal set of variables and operations manipulating these variables, which form an SCC in the PDG and exhibit a loop-carried dependence of distance 1. Furthermore, this SCC has no incoming edges from other SCCs in the PDG.

This definition exploits the fact that conditional expressions controlling loop exits will introduce control dependencies to every operation contained in the loop body. Since variables which are updated in every loop iteration are also of interest, this approach also looks for data dependencies in the other direction, i.e. from update operations in the loop body towards read operations in loop termination expressions. Together these two dependencies will form a loop-carried dependence cycle, or more generally an SCC in the PDG. Other operations may depend on this SCC, but it is the dominant SCC of operations, which does not depend on any other operations and variables that determines loop termination and, thus, constitutes the loop iterator.

In counted or affine loops the conventional iterator is intuitively covered by the definition given here, as it is this variable (often `i`) and its updates and checks that form the dominant SCC controlling execution of the remaining loop operations, which in turn form the loop "payload". Similarly, iterators of pointer-chasing loops are covered by the same definition as pointer updates and checks introduce a cyclic dependence relation on which the remaining loop body depends. However, Definition 4.3.1 also covers STL-like iterators, which are updated and checked in every single loop iteration, possibly involving calls to methods in other classes, though, which necessitate the use of inter-procedural analysis for their identification.

Abstraction and generalization of the properties of a loop iterator in the definition above now allows the development of an iterator recognition analysis operating on the compiler IR, thus enabling a source-language agnostic approach.

4.3.2 Static Analysis

The analysis for determining loop iterators presented here involves three stages closely following Definition 4.3.1:

1. **PDG construction**. The IR is assumed to provide a control flow graph (CFG) in static single assignment (SSA) form (Figure 4.7a). The algorithm from [26]

is applied to construct the control dependence graph (CDG) of the loop (Figure 4.7b). Additionally, the implicit def-use chain present in the intermediate representation is combined with a static dependence analysis of memory accesses based on [34] to build the data dependence graph (DDG) of the loop (Figure 4.7c). The PDG is produced by combining the CDG and DDG (Figure 4.7d).

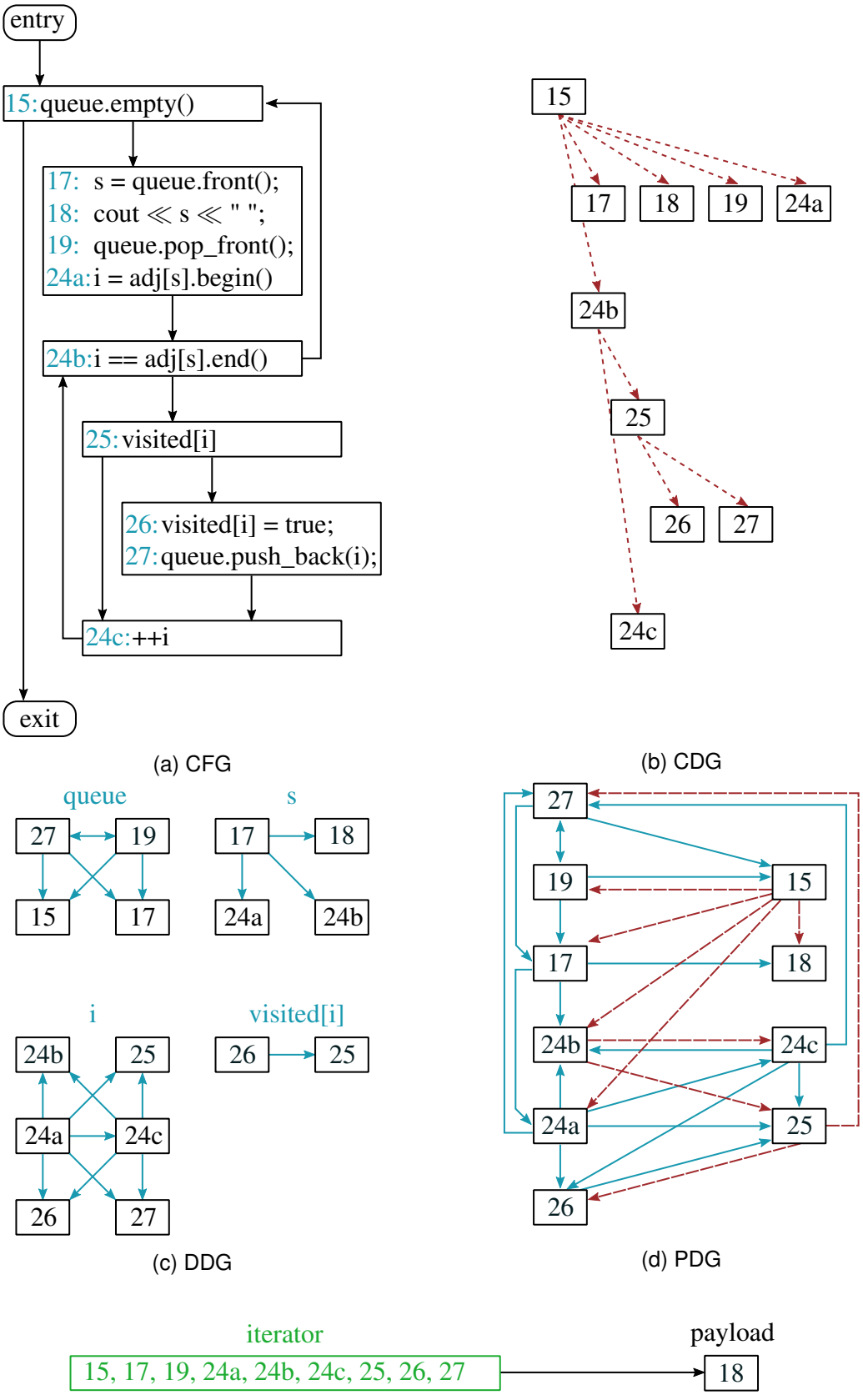
2. **Determine SCCs.** Once the program dependence graph of a loop is constructed, its strongly connected components are determined. A directed acyclic graph (DAG) connecting the SCCs is built using Kosaraju's algorithm [5].

3. **Dominant SCC and iterator recognition.** Finally, the dominant SCC, i.e. the one that has no incoming edges in the SCC DAG, is taken. This dominant SCC represents the loop iterator and the algorithm labels instructions represented by the SCC as "iterator instructions" and variables involved as "iterator variables" (Figure 4.7e). Inspection of the properties of these iterator instructions and variables reveals that together they satisfy Definition 4.3.1 by construction, showing that the loop iterator has indeed been recognised. Figure 4.8 illustrates an example labeling of the IR nodes of a loop extracted from the SPEC CPU2006 benchmarks.

4.3.3 Incorporating Profiling Information

Conservatism of the static analysis used as part of the construction of the DDG in section 4.3.2 is a limiting factor. *May* dependencies introduce spurious dependence relations, which may not materialize for any program execution on valid input data. This section investigates how incorporating profiling information obtained from instrumentation and execution of the target program can be used to improve iterator recognition. The profiling framework described in Chapter 3 is used to capture data dependencies.

Clearly, any approach relying on profiling information for the computation of data dependencies is prone to errors such as missing a dependence, which did not materialize in a specific execution trace, but could well occur in another [97]. However, there is still benefit in incorporating such unsafe information for two reasons: (a) an upper bound of the available dependencies can be determined, which enables the quantification of the scope for improvements of static analyses, and (b) some transformations, e.g. speculative parallelisation [76], are inherently resilient against data dependence violations and can benefit from more aggressive loop transformations.



(e) Strongly connected components (SCCs) of the PDG. The green SCC has no incoming edges and constitutes the iterator.

Figure 4.7: CFG, CDG, DDG, PDG and SCC for the `while`-loop in Fig. 4.4.

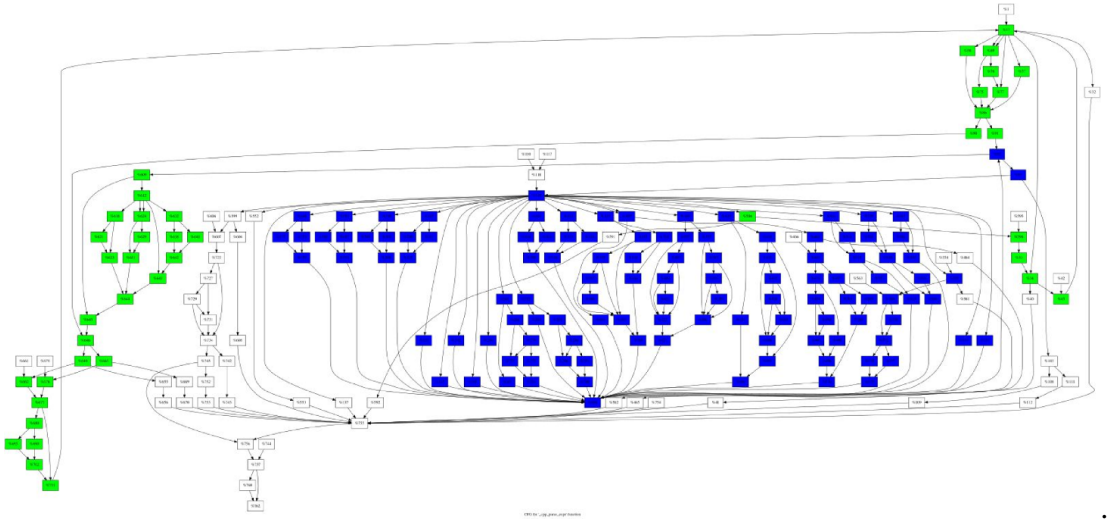


Figure 4.8: Illustration of the labeling of the CFG of a loop extracted from the SPEC CPU2006 benchmarks. The nodes in green are the iterator instructions, while the ones in blue are the payload. The remaining nodes are part of the CFG of the function containing the loop.

In principle, the PDG construction algorithm (stage 1 above) is complemented with profiling information similar to [43], i.e. for each of the statically detected *may* data dependencies the technique presented here refers to profiling information to resolve this *may* dependence as either *must* dependence or *no* dependence, based on whether or not a dependence was observed in the execution profile. The remaining stages of the algorithm remain unchanged.

Users are expected to profile their applications using representative, but reduced data sets, e.g. [27], and to focus their efforts on those parts of the program relevant to their targeted transformation in order to avoid excessive profiling costs.

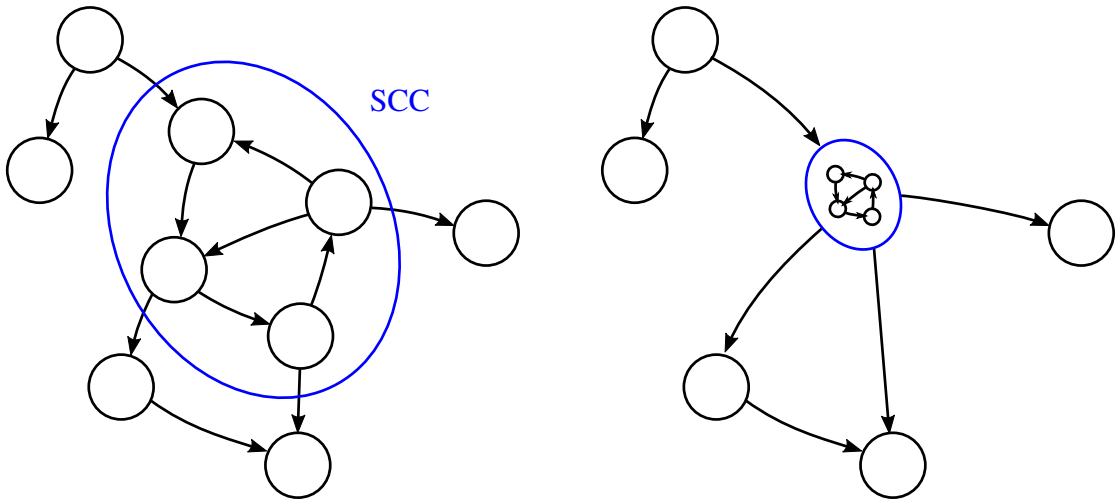
4.3.4 Implementation

4.3.4.1 LLVM Implementation

A prototype of the technique presented here has been implemented as an analysis pass in the LLVM compiler infrastructure. This allows one to analyse all of the SPEC CPU2006 benchmarks, since there are LLVM front-ends available for C/C++ (clang) and Fortran (flang). The LLVM IR is a CFG in SSA form as needed by the technique. Standard LLVM analyses allow for detecting loops in this graph and promoting memory accesses to virtual registers and thus simplifying the IR.

4.3.4.2 Instrumentation & Profiling

In order to collect memory access information about a subject program, the context aware memory profiler developed in Chapter 3 is applied. This begins with building a complete call graph of the program and computing context offsets for function calls. When these context offsets are accumulated for a given stack trace (i.e. a sequence of function calls) the result is always a context identifier that is unique for the given stack trace. Groups of recursive functions and indirect function calls are handled. The former is done by reducing the call graph to an SCC DAG and assigning a unique context ID to every SCC: the calls within the recursive groups have offsets of zero (see Figure 4.9). The latter – by adding calls to the runtime that indicate indirect function calls and function returns and keeping a stack of indirect calls at runtime.



(a) A recursive call graph.

(b) Contracted SCC DAG.

Figure 4.9: Handling recursion robustly. The strongly connected component in Figure 4.9a is represented by a single node in the contracted graph in Figure 4.9b. The resulting graph does not contain any further cycles, i.e. it is a DAG.

Once this map from function calls to context offsets is computed, each call is instrumented with calls to the runtime to advance and then the context ID is restored. Once this is done, all memory accesses are instrumented with a call to a runtime function that records the instruction ID and the address that is accessed.

Lastly, loops are also instrumented, since the analysis needs to know which iterations triggered a dependence and across which loop nesting level did the dependence occur. For this, loop entering and loop exiting edges in the CFG are instrumented, as

well as loop back-edges.

Once instrumentation is complete, the program is executed and a profile that consists of a list of the dependencies and the code coverage is collected. Each dependence is described by its

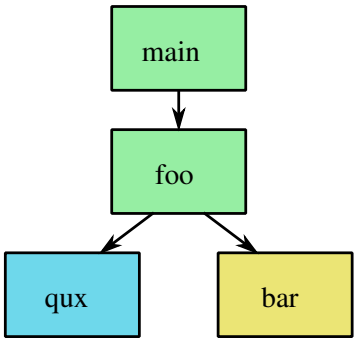
1. Type (RAW, WAW, WAR);
2. Source and target instructions;
3. Source and target contexts; and
4. List of loop iteration counters.

Iteration counters are obtained by counting the number of times a back-edge was encountered, rather than relating to the complex notion of iterators that the analysis presented in this chapter aims to extract.

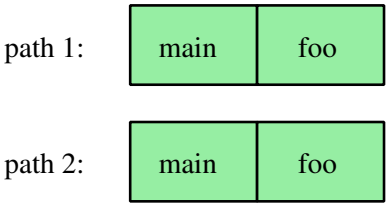
4.3.4.3 Incorporating Profiling Information

Once the profiling information is collected, it is incorporated back in the analysis to augment the static analysis results. Because unique contexts need to be mapped back to the functions in which they occur, the tree of all possible context IDs needs to be explicitly built. This is the call tree of the program (see Figure 4.10a). Each node of the call tree represents a runtime context: a specific sequence of call sites, starting with a call site in the main function. There is a unique context ID associated with each node. Each edge of the call tree represents a call site which might be encountered during the execution of the context that is represented by the source node of the edge. Note that a single function can appear many times in the call tree as the target of the last element in the sequence of call sites represented by a node. Similarly, a call site can appear many times as an edge in the tree: as often as the containing function is represented by a node.

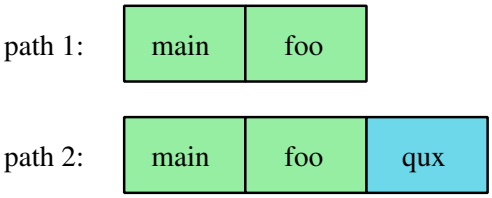
The size of the call tree is asymptotically exponential in the number of distinct functions in the program, so it is impossible to construct in an efficient manner. For this reason, a lazy tree access procedure is implemented, which builds only parts of the tree which are required to compute the nodes for context IDs which have actually been observed during the profiling run. The results are cached, so that the same part of the tree does not need to be computed twice. This puts a limit on the complexity of the analysis and ensures that the number of nodes in the tree that will be explicitly



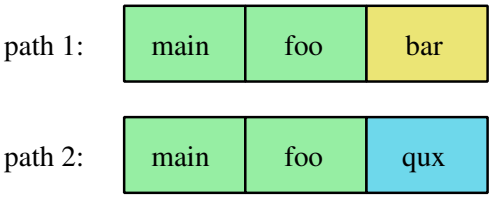
(a) The call tree of the program that profiling data is being incorporated for. Each node represents a context and each edge represents a function call in that context.



(b) The two paths are the same. The memory access instructions are in the same function (foo). They should be used themselves to record the dependence.



(c) One path is a prefix of the other. One memory access happens behind one or multiple function calls: in this case a call to qux. The top-level function call instruction should be used to record the dependence together with the memory access instruction that is on the same level in the call tree.



(d) The two paths share a prefix. There is a function which contains two different function call sites that eventually lead to the memory instructions that participate in the runtime dependence. Similarly to 4.10c, these call sites are used to record the dependence, instead of the instructions behind them.

Figure 4.10: Comparing the paths from the root of the call tree to the instructions in an observed dependence is necessary in order to decide which instructions to associate with the dependence. When the instructions are in the same runtime context (Figure 4.10b), then a dependence is constructed between them by the profiling information incorporation module. When the instructions do not happen to be in the same runtime context, the dependence should be constructed between a memory instruction and a function call instruction (Figure 4.10c) or two function call instructions (Figure 4.10d).

computed will be at most the number of contexts seen during the profiling run, which is less than exponential.

With the call tree built, for each dependence in the gathered profile, the paths from the root of the call tree to the context in which the source and target instructions of the dependence were encountered is computed. By taking the longest common prefix of these two paths the analysis finds the appropriate instructions to build the dependence between:

1. Between two memory instructions, if the paths are the same (the instructions are part of the same function: Figure 4.10b);

2. Between a memory instruction and a call instruction, if one path is a prefix of the other (one instruction happens behind a function call in the function in which the other instruction belongs to: Figure 4.10c);

3. Between two call instructions, if no path is a prefix of the other (the memory instructions happen behind different function calls within the same function: Figure 4.10d).

4.3.4.4 PDG Construction

Each of the dependencies encountered during profiling is added to the PDG built by static analysis, where *may* dependencies are treated as absent dependencies for the purpose of enabling aggressive, speculative transformation, but are re-inserted as encountered. In Section 4.4, the impact of resolving statically determined *may* dependencies which have not been covered by profiling, as either *must* dependence (=conservative lower bound) or *no* dependence (=aggressive upper bound) is discussed.

4.4 Evaluation

This section describes the experimental setup and the evaluation process of this chapter. It presents the results and offers an analysis explaining the stated observations.

4.4.1 Experimental Set-up

The methodology for iterator recognition and separation is evaluated against SPEC CPU2006 application benchmarks. All integer and floating-point benchmarks, covering codes written in C, C++ and Fortran are included. This is in contrast to e.g. [93]

where loop partitioning was applied to loops in selected functions, or [69] where small benchmarks kernel have been used for evaluation. In order to limit the time required for profiling, this section uses the `test` input data set, as no justified improvement from using the `ref` data set instead (see section 4.4.2.4) was discovered.

An LLVM implementation (version 3.9) of the technique as described in the previous section is used. For the iterator recognition pass two experiments are conducted: (a) the results rely on static analysis only for PDG construction, and (b) the analysis uses both static analysis and additional profiling information obtained from running an instrumented version of the benchmarks, where profiling information is fed back to the dependence analysis pass of the LLVM compiler. The host system uses four AMD Opteron 6376 CPUs (64 logical cores in total) and has 1TB of RAM available.

For each benchmark this section reports the total number of loops, the number of loops with affine iterators identified by Polly [35], the number of statically separable loops by the iterator recognition technique presented in this chapter, and the number of separable loops using additional profiling information.

The technique presented here is compared against Polly, since its scalar evolution based handling of induction variables and iterators can be considered state-of-the-art. Polly has been modified such that loops with affine iterators are counted even if the loop body violates additional constraints required for affine loops, e.g. affine array index functions. In particular, the comparison considers all loops where:

1. For each loop, there exists a single integer induction variable that is incremented from a lower to an upper bound by a constant stride.

2. Lower and upper bounds are affine expressions involving loop-invariant integer expressions and surrounding loop induction variables.

4.4.2 Results

The main results are presented in the table in Figure 4.1. For each benchmark the total number of loops and how many of them have affine iterators (=syntactically separable) is presented. This is compared to the generalized iterator recognition pass presented here when driven (a) by static dependence analysis and (b) profile-guided dependence analysis indicating lower and upper bounds, respectively. It can be observed that the novel iterator recognition pass can identify and separate substantially more loop iterators – for either programming language – than what is possible with affine iterator

Benchmark	Loops	Affine Iterators		Statically Separable		Separable After Profiling			
400.perlbench	1,333	364	27%	1,285	96%	1,292	97%	1,297	97%
401.bzip2	238	106	45%	191	80%	235	99%	238	100%
403.gcc	4,617	957	21%	4,581	99%	4,581	99%	4,588	99%
429.mcf	50	16	32%	50	100%	50	100%	50	100%
433.milc	426	257	60%	422	99%	424	100%	425	100%
445.gobmk	1,288	554	43%	1,272	99%	1,274	99%	1,281	99%
456.hmmer	876	337	38%	867	99%	867	99%	868	99%
458.sjeng	267	50	19%	265	99%	265	99%	267	100%
462.libquantum	98	36	37%	98	100%	98	100%	98	100%
464.h264ref	1,870	1,268	68%	1,859	99%	1,859	99%	1,869	100%
470.lbm	23	22	96%	23	100%	23	100%	23	100%
482.sphinx3	591	151	26%	582	98%	587	99%	587	99%
Total	$\Sigma(11,677)$	$\Sigma(4,118)$	$\varnothing(35\%)$	$\Sigma(11,495)$	$\varnothing(98\%)$	$\Sigma(11,555)$	$\varnothing(99\%)$	$\Sigma(11,591)$	$\varnothing(99\%)$
444.namd	623	450	72%	548	88%	557	89%	557	89%
447.dealII	7,323	4,115	56%	3,999	55%	4,239	58%	6,073	83%
450.soplex	759	360	47%	453	60%	532	70%	649	86%
453.povray	1,357	438	32%	975	72%	1,035	76%	1,220	90%
471.omnetpp	481	133	28%	162	34%	178	37%	233	48%
473.astar	119	42	35%	82	69%	107	90%	110	92%
483.xalancbmk	3,617	676	19%	1,907	53%	2,169	60%	2,712	75%
Total	$\Sigma(14,279)$	$\Sigma(6,214)$	$\varnothing(44\%)$	$\Sigma(8,126)$	$\varnothing(57\%)$	$\Sigma(8,817)$	$\varnothing(62\%)$	$\Sigma(11,554)$	$\varnothing(81\%)$
410.bwaves	85	84	99%	85	100%	85	100%	85	100%
416.gamess	21,386	19,970	93%	20,353	95%	20,386	95%	20,924	98%
434.zeusmp	547	533	97%	534	98%	537	98%	537	98%
435.gromacs (Fortran/C)	2,364	1,717	73%	2,088	88%	2,120	90%	2,273	96%
436.cactusADM (Fortran/C)	1,903	1,150	60%	1,523	80%	1,567	82%	1,724	91%
437.leslie3d	405	403	100%	403	100%	404	100%	404	100%
454.calculix (Fortran/C)	4,610	3,309	72%	3,877	84%	3,980	86%	4,409	96%
459.GemsFDTD	1,181	1,161	98%	1,165	99%	1,174	99%	1,178	100%
465.tonto	10,791	10,464	97%	10,610	98%	10,636	99%	10,710	99%
481.wrf (Fortran/C)	7,739	7,390	95%	7,568	98%	7,576	98%	7,618	98%
Total	$\Sigma(51,011)$	$\Sigma(46,181)$	$\varnothing(91\%)$	$\Sigma(48,206)$	$\varnothing(95\%)$	$\Sigma(48,465)$	$\varnothing(95\%)$	$\Sigma(49,862)$	$\varnothing(98\%)$

Table 4.1: Results for the SPEC CPU2006 benchmarks, grouped by programming language: C, C++, and Fortran.

recognition alone. Profiling information always increases the number of separable loops, in particular for the C++ benchmarks.

Results for > 75.000 loops across all SPEC CPU2006 applications, both integer and floating-point, are shown, grouped by programming language (C, C++ and Fortran). For each benchmark the total number of loops, the number of affine loops and their percentage of the total number of loops, the number and percentage of statically separable loops using the technique presented in this chapter, and the number and percentage of dynamically separable loops, are reported. Since profiling with standard data sets does not guarantee that a particular loop is executed, this report presents a *range*, i.e. a lower and an upper bound, for each benchmark for the dynamically separable loops. The lower bound corresponds to cases where *may* dependencies in non-profiled loops are conservatively approximated, whereas the upper bounds corresponds to a scheme where such unobserved *may* dependencies are resolved aggressively.

4.4.2.1 Comparison to Affine Loop Iterators

Inspection of the data in Figure 4.1 reveals that affine loop iterators are common in Fortran based programs and account for 90.5% of all loops in these applications, but are less frequently encountered in programs written in C and C++ (35.3% and 43.5%, respectively). However, even for C and C++ applications there exists great variance with individual programs, e.g. `458.sjeng`, `403.gcc` or `483.xalancbmk`, exhibiting only few affine iterators, whereas others including `470.lbm` make frequent use of such iterators.

Given that affine loop iterators are a prerequisite to the application of polyhedral loop transformations these results confirm that there is limited scope for such transformations on the SPEC CPU2006 C and C++ applications[2].

In contrast, the static analysis for iterator recognition and separation presented here is applicable to all benchmarks. For most C and Fortran programs almost all loop iterators can be separated from loop "payload", resulting in 94.5% and 98.4% of statically separable loops. Note that failure to separate iterators is *not* a failure of this technique, but an inherent loop property ("inseparable loop").

For C++ codes fewer separable iterators are observed, although this figure (56.9%) is still well above that for affine iterators (43.5%) and can be improved substantially using profiling information. We will see later that iterator separation for C++ appli-

[2] In recent work [15] some attempts have been made to extend the polytope model for general data-dependent control-flow.

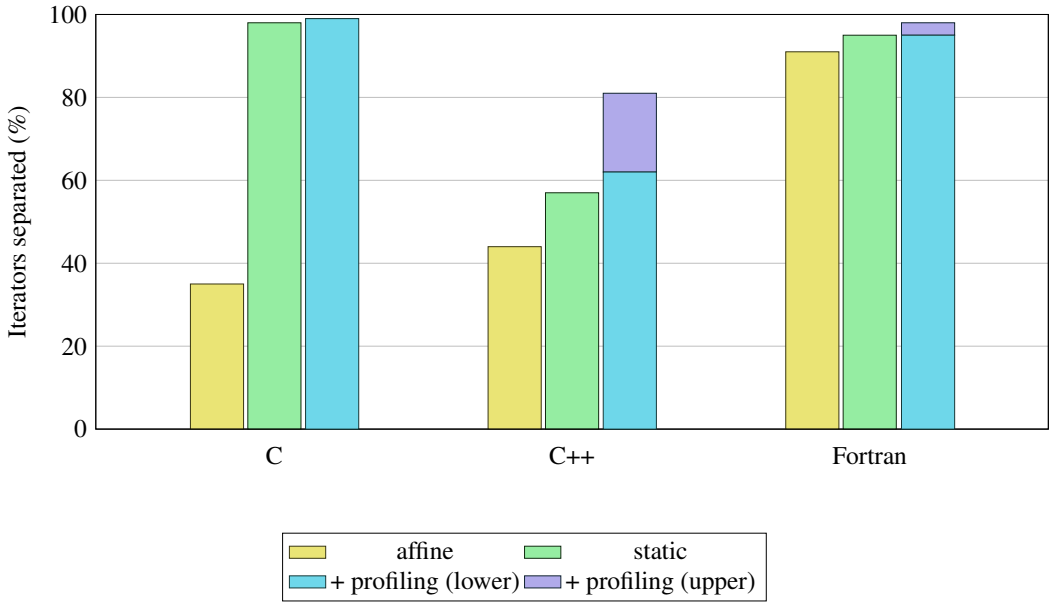


Figure 4.11: Breakdown of iterators recognized by affine, static and profile-guided analysis and further broken down by programming language.

cations can be vastly improved using profiling information, which suggests that static analysis is hampered by specific traits exhibited by C++ in general or by the specific set of applications in the benchmark suite, which is discussed in the following paragraph.

4.4.2.2 Evaluation By Programming Language

Again, consider the table in Figure 4.1 and also the chart in Figure 4.11. Static iterator recognition works well for Fortran and C programs with 94.5% and 98.4%, respectively, of all loops separable using static analysis alone. This is a slightly surprising result given that the C based SPEC applications tend to be more irregular and pointer based than their Fortran counterparts. However, we find that some C benchmarks such as `401.bzip2` and some mixed Fortran/C codes such as `436.cactusADM` and `454.calculix` have a lower than average number of statically separable loops. The majority of the unseparable loops in the mixed Fortran/C benchmarks are found in the part written in C.

The situation is different for the applications written in C++, where the average percentage of statically separable loops is substantially lower at 56.9% than for C or Fortran based codes. For some applications, e.g. `471.omnetpp` this percentage can be as low as 33.7%, whereas for `444.namd` 88.0% of its loops are statically separable. Lower separability figures for C++ applications can be attributed to following three reasons: (a) the C++ applications comprise fewer separable loops as can be seen when

comparing results to the upper bound obtained by profiling, and (b) C++ applications are harder to analyse statically using the analyses provided in the LLVM compiler. This later point can be validated by comparison against the lower bound of the profiling data, which suggests that even modest amounts and conservative use of dynamic information can lead to improvements for C++ programs over static analysis alone.

Some of the C++ applications in the SPEC CPU2006 suite contain a large number of non-natural and multi-exit loops [86], which defeat LLVM's static analysis. During the research prototyping related to this chapter it has also been observed that LLVM's ability to disambiguate accesses to fields and members in structures (and classes) is limited. For example, in `473.astar` it has been observed that LLVM's analyses conservatively report possible dependencies between array and other members contained in a class, where profiling information can help to disambiguate accesses.

4.4.2.3 Impact of Profiling Information

Profile information is complementary to static analysis and only adds to number of separable loops. It works best where static analysis is conservative and reports *may* dependencies, which do not materialize in actual program executions. However, for cases where static analysis already enables iterator separation for e.g. $> 98\%$ of all loops there is obviously limited scope for improvement (other than minimizing iterator instructions). This is the case for most Fortran and C benchmarks, although there are a number of notable exceptions. While only 191 out of 238 loops are statically separable for `401.bzip2` profiling enables separation of, at least, 235 loops, thus increasing recognition from 80.3% to 98.7%.

For C++ applications with their lower number of separable loops and iterators profiling improves separability by, on average, 5% (lower bound) and up to 24%. For `450.soplex`, for example, static analysis enables separation of 59.7% of all loops, whereas profiling contributes to an increase of greater than 10% and up to almost 26% depending on whether a conservative or aggressive scheme is used.

4.4.2.4 Coping with Limited Profiling Coverage

Table 4.2 presents the amount of loops executed during profiling the SPEC CPU2006 benchmarks using the `test` input set. Profiling with standard data sets does not guarantee loop coverage, i.e. some loops may not be executed during profiling. A breakdown of loops executed during profiling with SPEC CPU2006 `test` data sets is presented,

and it is compared to how many of these have affine iterators, or are statically separable or dynamically separable with the technique presented here, respectively.

Profiling inevitably incurs a substantial overhead and it is in the interest of the user to reduce the time for profiling. This can be achieved by using a smaller input data set, e.g. the `test` instead of the `ref` data set for SPEC CPU2006. However, a typical data set often does not fully exercise every code path, i.e. there is limited loop coverage. The `test` input data set for SPEC CPU2006 has been found to cover, on average, only 19% of all loops (see Figure 4.12a). As discussed in section 3.5.3, the `ref` input data set has also been used for eight (out of the 29) benchmarks for comparison. For four benchmarks the loop coverage did not change, for two others the increase was negligible (2 and 3 additional loops out of 297 and 329, respectively), and, surprisingly, for one of the benchmarks two loops fewer were covered. There was only one benchmark, 445.gobmk, which saw a significant increase in the coverage, raising from 23% to 73% of the total loops in the benchmark. Time for profiling, however, increased 185× on a geometric average over the eight benchmarks that were measured.

In general, loop coverage is important and in Figures 4.12b, 4.12c, and 4.12d we see that profiling information has a substantial impact on the ability to separate iterators, especially for the C++ applications. These findings suggest that profiling with standard data sets is not ideal, but a more targeted approach supported by techniques from the field of software testing, e.g. automated test case generation [9], should be considered, but this is beyond the scope of this chapter.

In practical terms end users would likely isolate code regions of interest, e.g. performance bottlenecks, and use targeted test harnesses to drive profiling specifically for these regions.

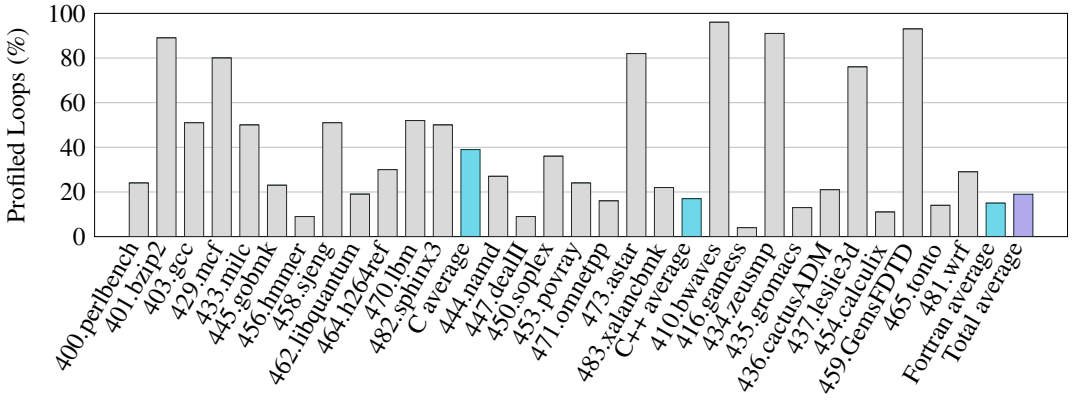
4.4.2.5 Evaluation of Iterator Size and Complexity

This section considers iterator size and complexity, i.e. what percentage of the instructions in a loop are part of the iterator and payload, respectively. This information is useful to evaluate the potential benefit of advanced parallelisation schemes as parallel speedup is related to the size of the payload, i.e. non-iterator code in the loop.

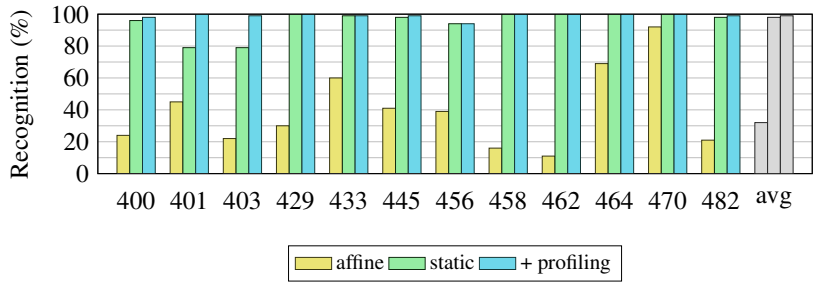
Consider the three diagrams in Figure 4.13, where the distribution of relative iterator sizes and their frequency across the SPEC CPU2006 benchmarks is plotted, broken down by programming language (C, C++, Fortran). There are two interesting observations: (a) Iterator sizes are not uniformly distributed, and (b) the distribution of iterator

Benchmark	Profiled Loops		Affine Loops		Statically Separable		Dynamically Separable	
400.perlbench	319	24%	76	24%	305	96%	312	98%
401.bzip2	211	89%	94	45%	167	79%	211	100%
403.gcc	2349	51%	506	22%	2336	99%	2336	99%
429.mcf	40	80%	12	30%	40	100%	40	100%
433.milc	213	50%	127	60%	211	99%	211	99%
445.gobmk	292	23%	120	41%	287	98%	289	99%
456.hmmer	79	9%	31	39%	74	94%	74	94%
458.sjeng	136	51%	22	16%	136	100%	136	100%
462.libquantum	19	19%	2	11%	19	100%	19	100%
464.h264ref	554	30%	383	69%	554	100%	554	100%
470.lbm	12	52%	11	92%	12	100%	12	100%
482.sphinx3	298	50%	64	21%	291	98%	296	99%
Total	Σ(4522)	∅(39%)	Σ(1448)	∅(32%)	Σ(4432)	∅(98%)	Σ(4490)	∅(99%)
444.namd	167	27%	112	67%	149	89%	158	95%
447.dealII	676	9%	368	54%	345	51%	585	87%
450.soplex	273	36%	113	41%	148	54%	227	83%
453.povray	329	24%	64	19%	225	68%	285	87%
471.omnetpp	77	16%	5	6%	27	35%	43	56%
473.astar	98	82%	29	30%	66	67%	91	93%
483.xalancbmk	810	22%	107	13%	325	40%	587	72%
Total	Σ(2430)	∅(17%)	Σ(798)	∅(33%)	Σ(1285)	∅(53%)	Σ(1976)	∅(81%)
410.bwaves	82	96%	81	99%	82	100%	82	100%
416.gamess	902	4%	796	88%	820	91%	853	95%
434.zeusmp	496	91%	489	99%	490	99%	493	99%
435.gromacs (mixed Fortran/C)	297	13%	205	69%	251	85%	283	95%
436.cactusADM (mixed Fortran/C)	408	21%	208	51%	311	76%	355	87%
437.leslie3d	309	76%	307	99%	307	99%	308	100%
454.calculix (mixed Fortran/C)	510	11%	269	53%	364	71%	467	92%
459.GemsFDTD	1094	93%	1082	99%	1084	99%	1093	100%
465.tonto	1488	14%	1437	97%	1446	97%	1473	99%
481.wrf (mixed Fortran/C)	2235	29%	2192	98%	2210	99%	2218	99%
Total	Σ(7821)	∅(15%)	Σ(7066)	∅(90%)	Σ(7365)	∅(94%)	Σ(7625)	∅(97%)

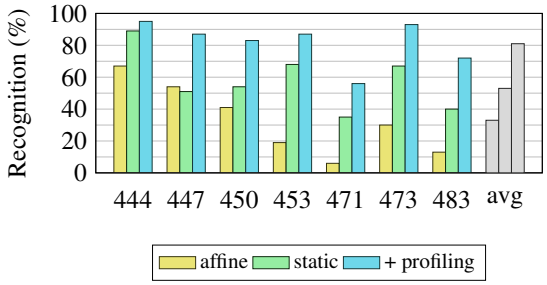
Table 4.2: Loops executed during profiling with SPEC CPU2006 `test` data sets and the coverage percentage per benchmark.



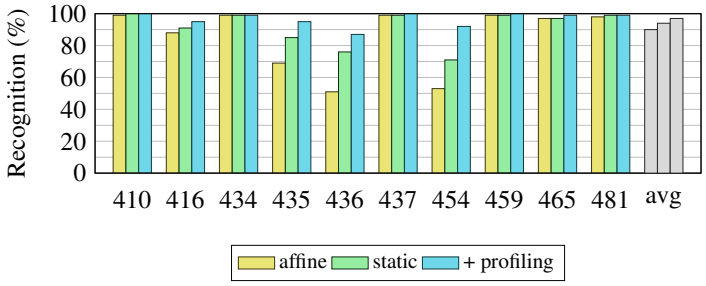
(a) Percentages of loops covered by the SPEC CPU2006 `test` data set out of all loops.



(b) Percentages of separated iterators from covered loops of C benchmarks.



(c) Percentages of separated iterators from covered loops of C++ benchmarks.



(d) Percentages of separated iterators from covered loops of Fortran benchmarks.

Figure 4.12: Profiling with the SPEC CPU2006 `test` data set does not cover all loops (a), yet even limited profiling information improves on static analysis alone for programs written in either programming language (b, c, d). Colours correspond to Table 4.2. Note that Figure 4.12(a) is a repeat of Figure 3.13 from page 77.

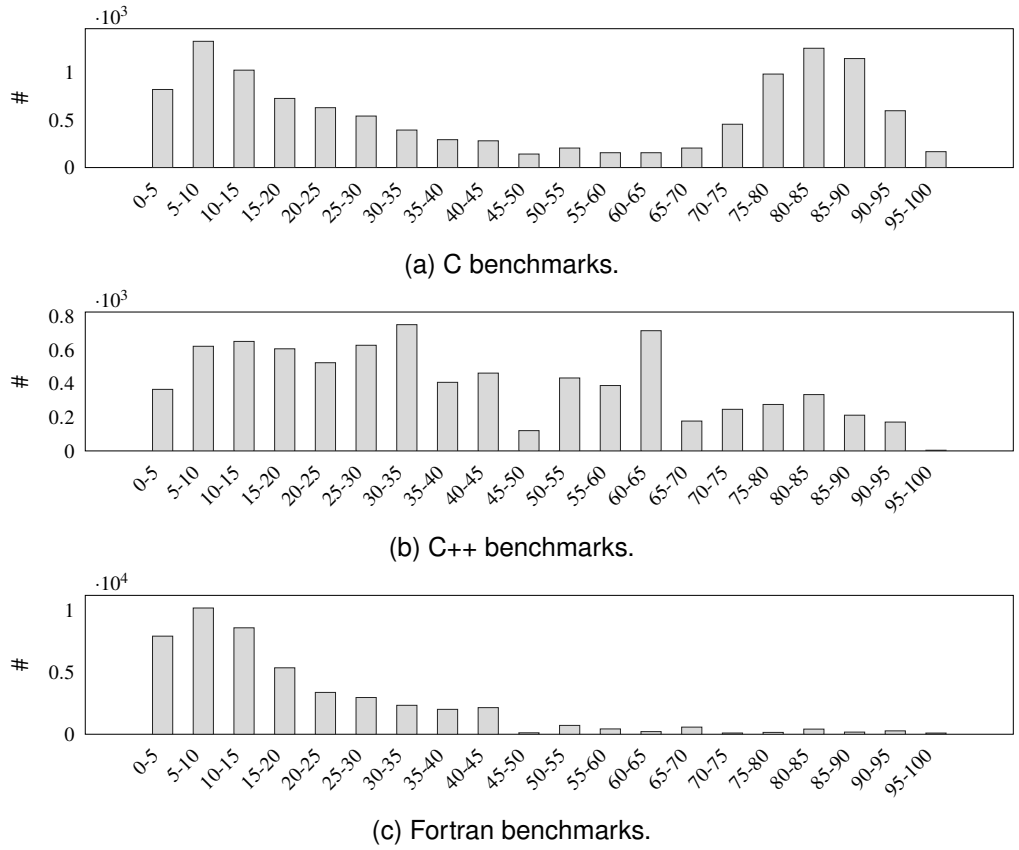


Figure 4.13: Distribution of relative loop iterator sizes as percentages of the total number of IR instructions in a loop (0% = iteratorless, 100% = inseparable) across all loops and SPEC CPU2006 benchmarks, broken down by programming language (C, C++, Fortran).

sizes varies significantly for the three programming languages used in the benchmark suite.

For the C benchmarks a bimodal distribution of iterator sizes is observed, where iterators are either very small (around 5-10% of loop instructions) or large (around 85% of loop instructions). The situation is different for Fortran codes, where the vast majority of iterators is small. The C++ applications exhibit the same bimodal trait as the C codes, but the peaks at the lower and higher end of the scale are less distinct.

Further inspection of the iterators reveals that small iterators are typically affine or near-affine iterators, which only require a few instructions to update and compare. Larger iterators are often complex and comprise additional control flow. Given that the Fortran benchmarks contain substantially more affine iterators, these dominate the distribution as expected. For the C and also C++ benchmarks, however, additional complex iterators similar to the motivating example in Figure 4.4 are observed.

4.4.2.6 Analysis and Profiling Overhead

Table 4.3 presents the runtimes of static analysis and dynamic analysis for the benchmarks in the SPEC CPU2006 suite. Note that the data is the same as that in Figure 3.12 on page 75. The time for static analysis increases with the number of IR instructions that are processed and the number of dependencies that are computed. The time for profiling depends on the behaviour of the respective benchmark for the given input set: in this case the `test` input set for SPEC CPU2006. In general, profiling incurs a substantial overhead and its use should be targeted at otherwise hard-to-analyse loops to avoid excessive profiling runs.

Static analysis needs to consider pairwise all instructions contained in a loop for dependence testing (using LLVM's DependenceAnalysis pass), resulting in $O(n^2)$ complexity. However, for most of the SPEC CPU2006 applications analysis is reasonably fast and only adds a few seconds to the overall compilation time. Figure 4.14 shows that in practice, the runtime of static analysis is roughly linear in the number of IR instructions (a) and in the number of static dependencies (b). A notable exception is the `416.gamess` application, which with its 2M+ IR instructions and 17.5M+ static dependencies spread over 21k+ loops, takes almost 11 minutes to analyse statically. For an application, which contributes almost 30% of all loops of the entire benchmark suite this is still acceptable. None of the C or C++ benchmarks takes longer than 30 seconds to analyse, though, and most of them can be processed in under 10 seconds.

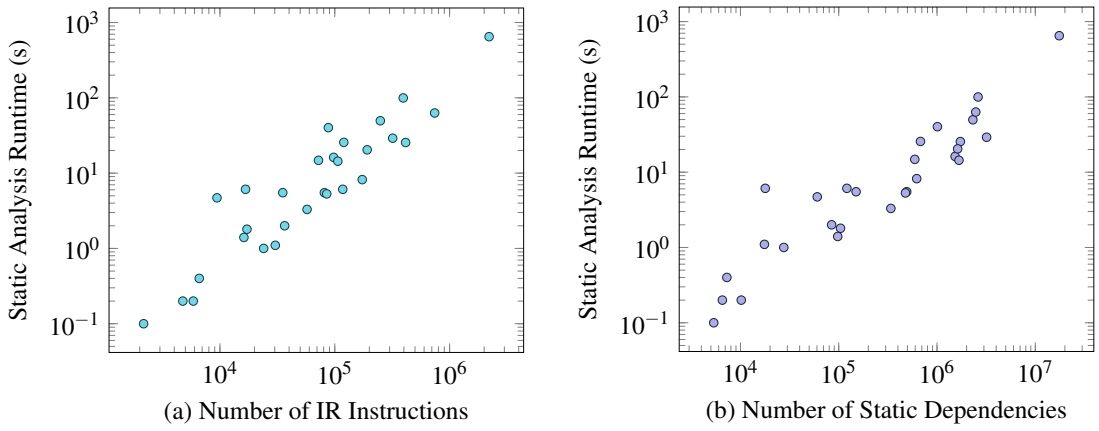


Figure 4.14: Runtime of static analysis versus number of IR instructions and number of static dependencies. In both cases, the relationship is roughly linear. Colours correspond to Table 4.3.

Benchmark	Loops	IR Instructions	Static Dependencies	Static Analysis	Profiling
	#	#	#	hh:mm:ss.s	hh:mm:ss.s
400.perlbench	1,333	97,762	1,527,084	00:00:16.2	00:17:57.0
401.bzip2	238	17,189	104,282	00:00:01.8	05:34:57.0
403.gcc	4,617	319,436	3,196,412	00:00:29.1	01:04:06.0
429.mcf	50	2,162	5,349	00:00:00.1	01:20:36.0
433.milc	426	30,347	17,535	00:00:01.1	**11:00:20.0**
445.gobmk	1,288	106,365	1,672,678	00:00:14.4	**20:17:17.0**
456.hmmer	876	57,373	339,107	00:00:03.3	06:20:00.0
458.sjeng	267	16,165	97,631	00:00:01.4	04:57:06.0
462.libquantum	98	4,750	6,542	00:00:00.2	00:05:26.0
464.h264ref	1,870	191,581	1,621,450	00:00:20.4	**34:10:49.0**
470.lbm	23	6,602	7,276	00:00:00.4	02:31:36.0
482.sphinx3	591	24,012	27,595	00:00:01.0	01:41:32.0
444.namd	623	81,184	491,392	00:00:05.5	09:48:56.0
447.dealII	7,323	413,968	1,728,428	00:00:25.5	**24:38:52.0**
450.soplex	759	36,595	84,633	00:00:02.0	00:09:51.0
453.povray	1,357	85,123	477,529	00:00:05.3	00:50:18.0
471.omnetpp	481	16,700	17,866	00:00:06.1	00:20:08.0
473.astar	119	5,854	10,188	00:00:00.2	06:29:47.0
483.xalancbmk	3,617	117,376	120,983	00:00:06.1	08:11:15.0
410.bwaves	85	9,421	60,430	00:00:04.7	06:38:13.0
416.gamess	21,386	2,208,496	17,534,550	**00:10:48.6**	00:28:21.0
434.zeusmp	547	72,157	594,104	00:00:14.8	**16:23:45.0**
435.gromacs (Fortran/C)	2,364	173,521	621,327	00:00:08.2	00:26:20.0
436.cactusADM (Fortran/C)	1,903	119,830	678,634	00:00:25.6	01:30:56.0
437.leslie3d	405	35,235	150,150	00:00:05.5	**33:01:49.0**
454.calculix (Fortran/C)	4,610	249,255	2,316,193	00:00:49.6	00:01:56.0
459.GemsFDTD	1,181	87,946	1,009,351	00:00:40.2	03:06:54.0
465.tonto	10,791	741,668	2,477,594	00:01:02.9	00:43:07.0
481.wrf (Fortran/C)	7,739	394,779	2,615,447	00:01:39.7	04:36:07.0

Table 4.3: Comparison of the time taken by static analysis and the time taken by profiling different benchmarks in the SPEC CPU2006 suite.

Profiling naturally incurs a much greater overhead than static analysis. For example, profiling of the SPEC CPU2006 applications using the `test` data sets requires several minutes and up to several hours as detailed in Table 4.3. With this framework, the overhead resulting from instrumented execution of a program yields, on average, a 760× slowdown. This means that for every second of non-instrumented execution the instrumented version of the same program will take around thirteen minutes to execute. This is clearly prohibitive in a continuous edit-compile-test cycle or on very large data sets. However, these are not the envisaged use cases of the profiling technique. Instead, the preferred application - as enabler of one-off transformations such as DSWP, source code rejuvenation or commutativity analysis supporting parallelisation - is inherently more tolerant to the observed profiling overhead in exchange for greater accuracy, especially on some complex C++ applications.

4.5 Related Work

Loop concepts and iterators are as old as the oldest high-level programming languages [32] and have received attention, in particular, in polyhedral loop analysis and programming language design. Loop identification [83] is the general problem of finding loops in programs. It is often based on Tarjan's interval-finding algorithm and is an essential step in performing various loop optimizations and transformations, but it is not concerned with identifying loop iterators.

Decidability of termination of several variants of simple integer loops, without branching in the loop body and with affine constraints as the loop guard (and possibly a precondition) has been considered in [14]. Whilst this work is related to iterator recognition it follows a decision theoretical approach. Reducible and irreducible loops are the subject of [38].

Efficient symbolic analysis of chains of recurrences supporting induction recognition is presented in [94]. Pointer-based array traversals are analysed and transformed to closed form array expressions in [30].

Static analysis is employed in [25] to determine loop iteration counts using polytope-based loop evaluation and program slicing. A constraint based approach to recognition of reductions is presented in [33], where a wide class of reductions including their loop iterators is recognized in the LLVM framework. However, generalized iterators as presented in this chapter are beyond the scope of their work.

HELIX [21, 69] is a speculatively parallelizing compiler, which would benefit from

iterator recognition. While HELIX applies parallelizing loop transformations, it relies on normalizable loops (equivalent to `while` loops), but it does not attempt to separate out loop iterator code. Instead, HELIX monitors *all* loop carried data dependencies without further distinction.

In [82] automatic parallelisation of loops that iterate over user-defined containers that have interfaces similar to the lists, vectors and sets in the Standard Template Library (STL) is demonstrated. However, this approach relies on the user inserting OpenMP directives into a serial program and, effectively, marking up loop iterators.

Partitioning of heap-allocated data structures and transformation of pointer-manipulating programs is a concern for high-level synthesis supporting FPGA design flows. Separation logic is used in [98] for the static analysis driving source-to-source transformation enabling loop parallelisation of programs comprising dynamic data structures and pointer-based memory accesses.

4.6 Further Analyses

Section 4.1 introduced some possible uses of the iterator recognition algorithm: decoupled software pipelining, code rejuvenation, and commutativity analysis. This section gives more detail of how iterator recognition can improve these techniques.

4.6.1 Decoupled Software Pipelining

Decoupled Software Pipelining (DSWP, [75]) is a technique powered by a static analysis, similar to the one presented in this chapter. Firstly, the program dependence graph of the CFG of a loop is constructed, and then the strongly connected components of this PDG are taken. In the analysis presented in this chapter the top-most dominant SCC is recognised as the iterator of the loop and the rest as the payload. In comparison, DSWP performs a load balancing technique, after evaluating the execution cost of each SCC, with the aim of achieving an equal separation of the runtime of the loop. DSWP then uses the parts separated in this way as the stages of a software pipeline. All of these steps are performed statically, and as such the grouping into SCCs is overly conservative: in the sense that it discovers fewer SCCs than there actually are (as shown by the authors of DSWP); and the load balancing algorithm is possibly inaccurate (especially in the case of input-dependent control flow inside the loops).

In [93], Vachharajani et al. propose adding speculation to DSWP in order to violate

rarely occurring dependencies and produce more fine-grain SCC decomposition of the PDG of the loop. This is an aggressive approach that alleviates the first of the problems – the prohibitively small number of strongly connected components – but selection of the pipeline stages is still done based on a static cost analysis, which can be highly inaccurate.

The technique presented in this chapter can be used in place of speculation to augment DSWP. In particular, Section 4.4 demonstrates that the addition of a profile driven data dependence analysis can increase the iterator detection by as much as 32% of the total amount of profiled loops (from 40% to 72%) for a single benchmark (483.xalancbmk) and by 6% (from 89% to 95%) across all loops. This leads to the conclusion that it will have the same or greater effect on the detection of pipeline stages for DSWP: in addition to separating loops that are statically inseparable, loops that are statically separable might end up with more stages due to the reduced number of dependencies reported by the dynamic analysis stage.

Using profiling information is more aggressive than static analysis, but also more conservative than using speculation: static *may* dependencies that are violated have never been encountered during the profiling stage. In comparison, speculation uses a dependence probability threshold. If this probability is, for example, five percent, then one in every twenty dependence speculations will result in a mis-speculation and increased runtime overhead. Contrary to speculation, using profiling information does not directly add any runtime overhead to the subject program.

Another advantage of using profiling over speculation is that since the framework is already in place and it has been decided that the time for profiling is not prohibitively large for the desired optimization, the profiling framework can easily be extended to include runtime measurements of parts of the loop. This will result in a more accurate runtime cost estimation of the different SCC of the CFG of the loop body and more balanced separation during the pipelining stage, especially if the loop contains data driven/dynamic control flow.

4.6.2 Code Rejuvenation

Source code rejuvenation is a relatively recent concept, introduced by Pirkelbauer et al. [77]. In contrast with refactoring, source code rejuvenation is used as a one-off transformation that automatically, or semi-automatically updates the source code of a legacy system so that it uses idioms and language features introduced by a new evo-

lution of the programming language that it is written in. Notable work in the field includes the 'demacrofier' introduced by Kumar et al. [50, 51]. The tool uses a classification of different C-style macros used in a C++ program in order to detect the ones that can be replaced with one of the following: a constant expression declaration, a lambda function, a function template, or an alias declaration. The time it takes for a complete transformation is in the order of magnitude of several hours.

Another important work is that of Wright et al. [99]. The authors present an automatic approach to minimizing the 'API surface' of their code-base: ensuring that the usage of old APIs is replaced with the usage of new ones. The system is highly parallelised and is configured so that it allows distributed analysis and transformation. A specific use case example is described, where 45,000 calls to a string splitting function were successfully migrated to use an updated interface. A specific runtime for that experiment is not reported, but it is stated that the system 'allows complex transformations across millions of lines of C++ code in a manner of minutes'. This is due to the massive parallelisation of the framework.

Mooij et al. propose using domain-specific models for software rejuvenation in [66]. This work revolves around a one-year effort of a complex rewrite of an X-ray software, involving the replacement of an ad-hoc XML and C# front-end framework with an easier to maintain C++ code and off-the-shelf GUI components that fit better with the rest of the code-base (written in C++ as well). The length of the project is necessitated by the size and complexity of the legacy software and the size of the desired change, and the hybrid approach involving both manual and automated techniques. The authors point out that the information about the original code base is stored in four distinct sources: documentation, developers, code base, and runtime behaviour; not all of which can be automatically processed. The presented approach is to perform rejuvenation as a three step process: first, extract the valuable business logic from the available sources and store them in domain-specific models, abstracting away implementation details; second, transform these models into redesigned generation models; third, generate the redesigned software using the transformed domain-specific models. In none of these stages do the authors completely automatically approach the problem at hand, but instead they take a more pragmatic approach aiming to minimise the combined time of automation execution and tool development. In pure research the latter is generally ignored, but in industry it can be more cost effective to trade some of the tool development time for manual effort [68].

Oikonomopoulos et al. take a different approach and suggest binary rejuvenation

[74]. The motivation for this idea is manifold. In addition to hardware evolution, the authors point out that there are security implications stemming from using 'stale' binaries. Despite static linking becoming outdated, developers sometimes shy away from using dynamically linked libraries, mainly due to the quality of such libraries and their failure to preserve source or binary compatibility. Finally, sometimes programmers add fallback functions to deal with platform-specific functionality (e.g. intrinsic functions as presented in Chapter 2) and these fallback functions become inadvertently exposed even in binaries for architectures that support said functionality. To attack these problems the authors suggest focusing on two problems: equivalence (proving a replacement candidate code is equivalent to a rejuvenated version) and granularity (should fragments of the CFG be replaced, or whole functions and libraries). The paper is only foundational and does not formalise any particular algorithm, but proposes initial directions of research in the area.

A rejuvenation transformation, that can be enabled by the iterator recognition technique presented in this chapter and that is categorically different than these developments is envisioned. Following the iterator separation stage, the iterator part of a loop can be further analysed and properties can be inferred. Similar to the examples presented in Section 4.1, the nature of the structure that is traversed can be analysed and then it can be replaced with an idiom or an STL object that does the same thing in a more maintainable manner. Part of this analysis can be access pattern analysis – perhaps the elements of the underlying structure are accessed only once – or memory layout analysis: if the data structure is spread around in memory then it could possibly be a linked list or a graph structure; if it's contiguous, then it's maybe a vector or an array. Such transformations can improve both maintainability and performance.

4.6.3 Commutativity Analysis

The concept of commutative code blocks has been studied as early as [16]. After asserting the importance of parallel computing, Bernstein explains how the notion relates to that of commutativity. Then, the author formally specifies the memory access conditions that are necessary for parallel and commutative transformations. More recently, Dinaz et al. have presented separability-based commutativity [85] and Aleen and Clark have presented output-based commutativity [8].

Separability-based commutativity is based on an object-oriented programming model. The notion of separability from [85] is different than the one presented in this chapter,

and concerns the separability of the methods subject to analysis into two parts: object modifying section and invocation section. This is necessary because if the method is deemed commutative it can only be executed in parallel if accesses to the object are performed atomically: in essence, only the invocation part is executed in parallel. Dinaz et al. use symbolic computation of expressions in order to determine whether an operation is commutative or not. The applications they demonstrate their technique on are embarrassingly parallel programs, e.g. an n-body solver and a water molecule simulation.

Output-based commutativity attempts to abstract away direct memory layout comparisons. In [8], Aleen and Clark propose a framework where commutativity is equivalent to the identity of the output of the program before and after reordering the execution of the subject function. Even if the memory state is different for the two executions, they can still be labelled as commutative if all the users of this memory state result in the same output. This process can continue until a function affects the program output and then a last comparison is performed on that. The authors also propose using random interpretation [37] to significantly speed-up symbolic interpretation.

More recently, von Koch studies the application of commutativity analysis for the detection of algorithmic skeletons in [96]. The author proposes a hybrid static/dynamic analysis approach to detecting commutativity and then presents a formal definition for a range of algorithmic skeletons based on their notion of commutativity. The framework is incomplete, however, and a major point of improvement is enabling the analysis of loops. This is where iterator recognition can prove critically important.

After separating a loop into an iterator and payload parts, it is possible to extract the payload into a separate function. Commutativity analysis can then be performed on this function, by recording the inputs presented to it during an ordinary loop pass, and then replaying them in different orders. If the output of the execution of the loop is always the same, the loop can be labelled as commutative, and based on the skeleton definitions developed in [96], opportunities for parallel execution can be detected. This approach is currently investigated by other researchers at the University of Edinburgh and the implementation uses the iterator recognition prototype presented in this chapter.

4.7 Summary, Conclusions & Future Work

Exact knowledge of loop iterators is critical to many loop analyses and transformations, yet existing compiler techniques are limited in their ability to accurately recognize irregular or complex loop iterators.

This chapter developed a generalized approach to iterator recognition, which enables multiple uses including loop optimization, parallelisation and general loop rewriting. Static analysis is shown to work well for C and Fortran applications, but complex C++ code benefits from additional profiling information. The approach to iterator recognition presented in this chapter was shown to work in practice and the LLVM prototype implementation is was found to be capable of separating a substantially larger number of loops and iterators than previous techniques. This was based on an evaluation against the full set of applications contained in the SPEC CPU2006 suite. Future work will focus on integration of the iterator recognition technique presented here with advanced loop parallelisation techniques.

Chapter 5

Summary

This thesis has investigated the analysis of legacy source codes and methods for translating their functionality and performance to architectures that the program was not originally intended to run on. It presents three complementary techniques that advance such analysis: the automatic extraction of information encoded in the usage of target-specific SIMD intrinsic functions, the tracking and recording of runtime data dependencies, and the automatic separation of loops into an iterator and payload. This chapter summarises these contributions and the related experimental results, and discusses the limitations and potential improvements of the methodologies presented.

5.1 Contributions

5.1.1 Intrinsic Translation

Chapter 2 presented FREE RIDER: a novel methodology for taking advantage of target-specific intrinsic functions when retargeting a legacy software. The key idea is that instead of trying to de-optimise a program which uses such functions with the aim of making it portable again, the optimisation information encoded in the usage of the intrinsic function can be used in order to produce an optimised program for the target architecture 'for free'. This is done by employing a graph-based intermediate representation descriptions of the intrinsic functions available on the two platforms. Intrinsic functions and their surroundings in the source code are combined into a graph which is then covered with sub-graphs corresponding to target intrinsics. If there are any parts left uncovered, they are implemented using portable source code.

The experiments show that this methodology allows FREE RIDER to reach 96%

of the speedup that a human expert can achieve by manually retargeting the program, but without any of the added development overhead. The set of benchmarks used for this experiment consisted of eight sample programs from the OpenCV computer vision library, which are available in optimised form for both the source and target architectures. Another experiment was also performed, where the subject program – a UAV autopilot – was only available for the source architecture. The automatically retargeted version managed to achieve a speedup of 3.73 on a processor with a vector unit of width four.

5.1.2 Data Dependence Profiling

After showing that target-specific optimisations can be used to achieve high-performing retargetted versions of a legacy program, this thesis moves on to investigating the analysis of programs that do not have such optimisations in place. Static analysis is identified as a limiting factor in the success of achieving fast execution on multi-core architectures, and a data dependence profiling framework is developed.

Chapter 3 builds the theory for, and describes such a framework, with the goal of achieving completeness, fine granularity, and precise context tracking, over a range of different programs. For this reason the chapter tackles problems generally ignored in other profiling systems, in particular the treatment of indirect and recursive function calls while tracking the runtime context of a program. The framework is evaluated on the whole of the SPEC CPU2006 benchmark suite and manages to build data dependence profiles for each of the programs, achieving the completeness goal.

The granularity achieved by the profiler is fine: it builds a dependence graph between individual instructions for each function of the program. If a memory access that results in a dependence occurs behind a function call, a dependence is constructed using the corresponding call instruction rather than the memory access instruction from the other function. The recorded contexts include inter-procedural loop nesting information, as well as information about the call stack. The resulting data structure is exposed via the LLVM compiler infrastructure and as such is easily accessible from following compiler passes.

5.1.3 Iterator Recognition

The concept of a loop iterator is then presented. It is a generalisation of ideas like induction variables, C++ iterators, and recursive data structures used for traversal. In

a nutshell, a loop iterator is defined as the collection of variables and operations performed in a loop, that affect the decision of when the loop terminates. The other part of the loop, which might be empty, is called the 'payload' of the loop.

Chapter 4 compares different notions of concepts similar to loop iterator and then presents a technique based on dependence analysis for detecting loop iterators. The technique involves building a program dependence graph that combines control and data dependencies, extracting the strongly connected components of that graph, and identifying the top-most SCC as the loop iterator. The chapter discusses why this algorithm works and then applies it to the SPEC CPU2006 benchmarks in order to compare it with the state-of-the-art scalar evolution technique.

The approach is discovered to be able to detect the iterator of 98%, 57%, and 95% of the loops of the benchmarks written in C, C++, and Fortran respectively. The same result for the scalar evolution technique (which detects only 'affine iterators') is 35%, 44%, and 91%, respectively. The chapter further shows that adding dynamic dependence information the recognition can be increased to the ranges 99%–99%, 62%–81%, and 95%-98%. The uncertainty comes from the incomplete coverage of the amount of loops that are profiled.

If we look only at the loops that are exercised by the dynamic data profiling, affine loops are detected by scalar evolution in 32%, 33%, and 90% for C, C++, and Fortran, respectively. In contrast, the approach presented here can detect 98%, 53%, and 94% of the iterators statically, and 99%, 81%, and 97% of the iterators using dynamic information. This goes to show the importance of coverage when using dynamic profiling and Chapter 4 asserts that increasing coverage will provide more precise detection of more of the program loops.

5.2 Critical Analysis and Further Work

5.2.1 Limitations of Intrinsic Translation

The methodology presented in Chapter 2 focuses on intrinsic functions that provide access to the vector operations available on a processor. Other operations available via intrinsics are not supported, in particular operations that enable the control of the memory hierarchy or intrinsics for synchronisation of multiprocessing or atomic operations. This is partly due to limitations in the intermediate representation language. The choice of this language is not a critical part of the technique, and it can be modi-

fied to allow the expression of any special functionality shared by multiple platforms. The difficulty is in the implementation of this functionality in a portable way, in case a platform does not support it.

A more fundamental limitation is the fact that platforms can be categorically different. An optimisation that is beneficial on one platform might be cost-ineffective on another one. Ultimately, the methodology's strength is also its weakness: it can only translate optimisations that are already present in the source code; it cannot come up with new optimisations without the intervention of an expert.

5.2.2 Performance of Data Dependence Profiling

The framework for data dependence profiling presented in this thesis achieves instruction-level context-sensitive precision. It is robust enough to profile all of the SPEC CPU2006 benchmarks, with all of their idiosyncrasies. However, it is considerably slower and takes more memory than other frameworks for data dependence profiling. Innovations from these frameworks can be incorporated into the one presented in Chapter 3 in order to improve its runtime and memory performance.

In particular, parallelisation can be added to the profiling runtime. Multiple data accesses can be recorded at the same time in a pipelining fashion. The largest part of the execution time is spent in constructing a dependence when one is detected and updating the relevant structures. Since this is a multistep process it is appropriate for pipelining.

To simplify the development of the framework, traces and temporary files are in a human readable format. This results in an increase of the memory footprint and can be avoided if these information streams are encoded in a binary form. Also, applying a compression on the data can further reduce the memory demand and increase the domain of programs that can be profiled in practice.

5.2.3 Specialisation of Iterator Recognition

The approach presented in Chapter 4 outperforms the state-of-the-art scalar evolution technique in discovering the iterators of loops. This improvement is increased by adding dynamic data dependence information. However, it is possible that a loop can be separable for some inputs or some call sites and not separable for others. In such cases, it might be beneficial to duplicate the subject loop and call one of the versions in one case (for one type of call sites/input data) and the other in another case. This

will allow the separable version of the loop to be detected as such and create more optimisation opportunities. The complexity of this approach lies in the categorisation of data inputs/call sites.

Chapter 4 discussed potential applications of the iterator recognition analysis. One application is the improvement of decoupled software pipelining via the addition of dynamic data profiling information. This can potentially provide the benefit offered by adding speculation to DSWP, but without the added runtime overhead. This claim has not been verified, however.

Another use of iterator recognition is automatic source code rejuvenation. While research in this area is focusing on syntactic rewrite tools, like demacrofication or API rejuvenation, the technique presented here can potentially be used to enable an transformation which replaces ad-hoc loop iterators that can be recognised as idiomatic constructs with the respective idioms. The complexity of this research is in the development of such idiom recognition techniques.

A last application of iterator recognition is commutativity analysis. If the payload of a loop – the set of variables and operations which do not affect the iterator – can be executed in an order different than the original and produce the same results, then the loop payload can potentially be executed in parallel. There is already some research in this area, but it has not been successfully applied to loops yet. This thesis provides the necessary analysis that can support the extension of commutativity analysis to loops.

Bibliography

[1] International Technology Roadmap for Semiconductors, Executive Summary. Technical report, 2007.

[2] Flang. https://github.com/flang-compiler/flang, 2017.

[3] SPEC CPU 2017. https://www.spec.org/cpu2017/, June 2017.

[4] TIOBE Index. https://www.tiobe.com/tiobe-index/, July 2017.

[5] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1983.

[6] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[7] S. Aldea, D. R. Llanos, and A. González-Escribano. Using SPEC CPU2006 to evaluate the sequential and parallel code generated by commercial and open-source compilers. *The Journal of Supercomputing*, 59(1):486–498, Jan. 2012.

[8] F. Aleen and N. Clark. Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 241–252, New York, NY, USA, 2009. ACM.

[9] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, Aug. 2013.

[10] ARM Ltd. Cortex^{TM}-M4 Devices Generic User Guide. http://infocenter.arm.com/help/topic/com.arm.doc.dui0553-/, 2010.

[11] U. Banerjee. *Dependence Analysis for Supercomputing*. Springer, 1988.

[12] D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy. A new approach to DSP intrinsic functions. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 10 pp. vol.1–, Jan 2000.

[13] L. A. Belady and M. M. Lehman. A Model of Large Program Development. *IBM Syst. J.*, 15(3):225–252, Sept. 1976.

[14] A. M. Ben-Amram, S. Genaim, and A. N. Masud. On the termination of integer loops. *ACM Trans. Program. Lang. Syst.*, 34(4):16:1–16:24, Dec. 2012.

[15] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct. 1966.

[17] K. Bierhoff. Iterator specification with typestates. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, SAVCBS '06, pages 79–82, New York, NY, USA, 2006. ACM.

[18] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] S. Breselor. Why 40-Year-Old Tech is Still Running America's Air Traffic Control. Wired, https://www.wired.com/2015/02/air-traffic-control/, February 2015.

[20] P. Bright. Failed Windows 3.1 System Blamed for Shutting Down Paris Airport. Ars Technica, https://arstechnica.co.uk/information-technology/2015/11/failed-windows-3-1-system-blamed-for-taking-out-paris-airport/, November 2015.

[21] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. Helix: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[22] J. Ceng, W. Sheng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. Modeling instruction semantics in ADL processor descriptions for C compiler retargeting. *Journal of VLSI signal processing systems for signal, image and video technology*, 43(2-3):235–246, 2006.

[23] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew. Data Dependence Profiling for Speculative Optimizations. In *Compiler Construction*, Lecture Notes in Computer Science, pages 57–72. Springer, Berlin, Heidelberg, Mar. 2004.

[24] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, Oct 2004.

[25] D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *2009 International Symposium on Code Generation and Optimization*, pages 136–146, March 2009.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[27] V. Escuder and R. Rico. Reduced input data sets selection for SPEC CPUint2006. Technical Report TR-HPC-02-2009, Department of Computer Engineering, Universidad de Alcalá, Spain, April 2009.

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[29] K. Flinders. Big Banks' Legacy IT Systems Could Kill Them. http://www.computerweekly.com/news/2240212567/Big-banks-legacy-IT-systems-could-kill-them, January 2014.

[30] B. Franke and M. O'Boyle. Array recovery and high-level transformations for DSP applications. *ACM Trans. Embed. Comput. Syst.*, 2(2):132–162, May 2003.

[31] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, Jan. 1995.

[32] W. K. Giloi. Konrad Zuse's Plankalkül: The first high-level, "non Von Neumann" programming language. *IEEE Ann. Hist. Comput.*, 19(2):17–24, Apr. 1997.

[33] P. Ginsbach and M. F. P. O'Boyle. Discovery and exploitation of general reductions: A constraint based approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 269–280, Piscataway, NJ, USA, 2017. IEEE Press.

[34] G. Goff, K. Kennedy, and C.-W. Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 15–29, New York, NY, USA, 1991. ACM.

[35] T. Grosser, A. Groesslinger, and C. Lengauer. Polly – Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, Dec. 2012.

[36] S. Guelton. SAC: An Efficient Retargetable Source-to-Source Compiler for Multimedia Instruction Sets. http://www.cri.ensmp.fr/classement/doc/A-429.pdf, 2010.

[37] S. Gulwani. *Program Analysis Using Random Interpretation*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2005.

[38] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, July 1997.

[39] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[40] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 121–130, New York, NY, USA, 2010. ACM.

[41] A. Irrera. Banks scramble to fix old systems as IT 'cowboys' ride into sunset. http://uk.reuters.com/article/uk-usa-banks-cobol-idUKKBN17C0DZ, April 2017.

[42] W. Jiang, C. Mei, B. Huang, J. Li, J. Zhu, B. Zang, and C. Zhu. Boosting the Performance of Multimedia Applications Using SIMD Instructions. In R. Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 59–75. Springer Berlin Heidelberg, 2005.

[43] N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 148–159, Piscataway, NJ, USA, 2017. IEEE Press.

[44] A. Ketterlin and P. Clauss. Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 437–448, Washington, DC, USA, 2012. IEEE Computer Society.

[45] J. Kim. Context-Aware Memory Dependence Profiling. Master's thesis, Pohang University of Science and Technology, 2017.

[46] M. Kim, N. B. Lakshminarayana, H. Kim, and C. K. Luk. SD3: An Efficient Dynamic Data-Dependence Profiling Mechanism. *IEEE Transactions on Computers*, 62(12):2516–2530, Dec. 2013.

[47] M. H. Kim. A new iteration mechanism for the C++ programming language. *SIGPLAN Not.*, 30(1):20–26, Jan. 1995.

[48] G. Koharchik and K. Jones. An introduction to GCC compiler intrinsics in vector processing. Linux Journal, http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing, September 2012.

[49] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Int. J. Parallel Program.*, 28(4):347–361, Aug. 2000.

[50] A. Kumar, A. Sutton, and B. Stroustrup. The demacrofier. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 658–661, Sept. 2012.

[51] A. Kumar, A. Sutton, and B. Stroustrup. Rejuvenating C++ programs through demacrofication. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 98–107, Sept. 2012.

[52] M. Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Trans. Comput.*, 37(9):1088–1098, Sept. 1988.

[53] L. Lamport. The Parallel Execution of DO Loops. *Commun. ACM*, 17(2):83–93, Feb. 1974.

[54] W. Landi. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, Dec. 1992.

[55] J. R. Larus. Loop-Level Parallelism in Numeric and Symbolic Programs. *IEEE Trans. Parallel Distrib. Syst.*, 4(7):812–826, July 1993.

[56] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[57] V. Lipets, N. Vanetik, and E. Gudes. Subsea: an efficient heuristic algorithm for subgraph isomorphism. *Data Mining and Knowledge Discovery*, 19(3):320–350, 2009.

[58] S. Manilov, B. Franke, A. Magrath, and C. Andrieu. Free Rider: A Tool for Retargeting Platform-Specific Intrinsic Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, LCTES'15, pages 5:1–5:10, New York, NY, USA, 2015. ACM.

[59] S. Manilov, B. Franke, A. Magrath, and C. Andrieu. Free Rider: A Source-Level Transformation Tool for Retargeting Platform-Specific Intrinsic Functions. *ACM Trans. Embed. Comput. Syst.*, 16(2):38:1–38:24, Dec. 2016.

[60] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[61] L. Meier, P. Tanskanen, L. Heng, G. H. Lee, F. Fraundorfer, and M. Pollefeys. Pixhawk: A micro aerial vehicle design for autonomous flight using onboard

computer vision. *Autonomous Robots*, pages 1–19, 2012. 10.1007/s10514-012-9281-4.

[62] S. P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012.

[63] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1107–1116, Washington, DC, USA, 2013. IEEE Computer Society.

[64] T. Moertel. Tricks of the trade: Recursion to Iteration, Part 1: The Simple Method, secret features, and accumulators. http://blog.moertel.com/posts/2013-05-11-recursive-to-iterative.html, 2013.

[65] T. Moertel. Tricks of the trade: Recursion to Iteration, Part 2: Eliminating Recursion with the Time-Traveling Secret Feature Trick. http://blog.moertel.com/posts/2013-05-14-recursive-to-iterative-2.html, 2013.

[66] A. J. Mooij, G. Eggen, J. Hooman, and H. van Wezep. Cost-Effective Industrial Software Rejuvenation Using Domain-Specific Models. *Theory and Practice of Model Transformations*, 2015.

[67] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[68] R. Munroe. Automation. https://xkcd.com/1319/, January 2014.

[69] N. Murphy, T. Jones, R. Mullins, and S. Campanoni. Performance implications of transient loop-carried data dependences in automatically parallelized loops. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 23–33, New York, NY, USA, 2016. ACM.

[70] A. Murray and B. Franke. Compiling for automatically generated instruction set extensions. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 13–22, New York, NY, USA, 2012. ACM.

[71] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 151–160, Washington, DC, USA, 2011. IEEE Computer Society.

[72] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short SIMD architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[73] E. M. Nystrom, R. D.-C. Juy, and W.-M. W. Hwuz. Characterization of repeating data access patterns in integer benchmarks. In *Proceedings of the 28th International Symposium on Computer Architecture*, September 2001.

[74] A. Oikonomopoulos, C. Giuffrida, S. Rawat, and H. Bos. Binary Rejuvenation: Applications and Challenges. *IEEE Security Privacy*, 14(1):68–71, Jan. 2016.

[75] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[76] V. Packirisamy, A. Zhai, W.-C. Hsu, P. C. Yew, and T. F. Ngai. Exploring speculative parallelism in SPEC2006. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 77–88, April 2009.

[77] P. Pirkelbauer, D. Dechev, and B. Stroustrup. Source code rejuvenation is not refactoring. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '10, pages 639–650, Berlin, Heidelberg, 2010. Springer-Verlag.

[78] G. Pokam, S. Bihan, J. Simonnet, and F. Bodin. SWARP: a retargetable preprocessor for multimedia instructions. *Concurrency and Computation: Practice and Experience*, 16(2-3):303–318, 2004.

[79] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'05, pages 218–232, Berlin, Heidelberg, 2005. Springer-Verlag.

[80] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 444–448, New York, NY, USA, 1995. ACM.

[81] W. M. Pottenger. Induction variable substitution and reduction recognition in The Polaris parallelizing compiler. Master's thesis, University of Illinois at Urbana-Champaign, 1995.

[82] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. *Semantic-Driven Parallelization of Loops Operating on User-Defined Containers*, pages 524–538. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[83] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2):175–188, Mar. 1999.

[84] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[85] M. C. Rinard and P. C. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

[86] R. E. Rodrigues, P. Alves, F. Pereira, and L. Gonnord. Real-World Loops are Easy to Predict: A Case Study. In *Workshop on Software Termination (WST'14)*, Vienna, Austria, July 2014.

[87] K. Rupp. 40 years of microprocessor trend data. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/, June 2015.

[88] Y. Sato, Y. Inoguchi, and T. Nakamura. Whole program data dependence profiling to unveil parallel regions in the dynamic execution. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 69–80, Nov. 2012.

[89] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Program.*, 28(4):363–400, Aug. 2000.

[90] C. Tenllado, L. Piñuel, M. Prieto, F. Tirado, and F. Catthoor. Improving superword level parallelism support in modern compilers. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, pages 303–308, New York, NY, USA, 2005. ACM.

[91] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[92] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[93] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[94] R. Van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 118–132, London, UK, 2001. Springer-Verlag.

[95] R. Vanka and J. Tuck. Efficient and Accurate Data Dependence Profiling Using Software Signatures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 186–195, New York, NY, USA, 2012. ACM.

[96] T. J. K. E. von Koch. *Automated Detection of Structured Coarse-Grained Parallelism in Sequential Legacy Applications*. PhD thesis, University of Edinburgh, 2014.

[97] T. J. K. E. von Koch and B. Franke. Variability of data dependences and control flow. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 180–189, 2014.

[98] F. J. Winterstein, S. R. Bayliss, and G. A. Constantinides. Separation logic for high-level synthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 9(2):10:1–10:23, Dec. 2015.

[99] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. Large-Scale Automated Refactoring Using ClangMR. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 548–551, Washington, DC, USA, 2013. IEEE Computer Society.

[100] H. Yu and Z. Li. Fast Loop-level Data Dependence Profiling. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 37–46, New York, NY, USA, 2012. ACM.

[101] V. Zhislina. From ARM NEON to Intel SSE – the automatic porting solution, tips and tricks. Intel Developer Zone, http://software.intel.com/en-us/blogs/2012/12/12/from-arm-neon-to-intel-mmxsse-automatic-porting-solution-tips-and-tricks, February 2014.